Finite-State Techniques

Automata, Transducers and Bimachines

Stoyan Mihov and Klaus U. Schulz

November 29, 2018

Preface

Finite-state techniques provide theoretically elegant and computationally efficient solutions for various (hard, non-trivial) problems in text and natural language processing. Due to its importance in many fundamental applications, the theory of finite-state automata and related finite-state machines has been extensively studied and its development still continues.

This textbook describes the basics of finite state technology, following a combined mathematical and implementational point of view. It is written for advanced undergraduate and graduate students in computer science, computational linguistics and mathematics. Though concepts are introduced in a mathematically rigorous way and correctness proofs for all procedures are given, the book is not meant as a purely theoretical introduction to the subject. The ultimate goal is to bring students to a position where they can both understand and implement complex finite-state based procedures for practically relevant tasks. Some of the specific features of the book are the following:

1. The spectrum of finite-state machines that are covered is not restricted to classical finite-state automata and "recognition" tools. We also treat important "input-output" and "translation" devices such as multi-tape automata, finite-state transducers and bimachines. All these machines can be used, e.g., for efficient text rewriting, information extraction from textual corpora, and morphological analysis.

2. After a conceptual introduction, full implementations/executable programs are given for all procedures, including a documentation of the programming code. In this way it is possible to observe the concrete behaviour of algorithms for examples selected by the readers. It is also not difficult to enhance the given programs by means of self-written trace functionalities, which helps to even better understand particular details of the algorithms and programs presented.

3. Conceptual descriptions and implementations in a natural way lead to an intermediate goal reached at the end of the book. Here we show how to use the technology introduced for solving some fundamental practical problems: the implementation of large lexica as deterministic finite-state automata, procedures for pattern search in strings, and flexible efficient procedures for matching-based text rewriting.

Students can use the techniques and programs presented as a starting point for realizing more advanced own applications, e.g., in the field of lexica, knowledge

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

bases, search and information extraction.

It should be mentioned that programs presented are "pedagogical" in the sense that the main focus is transparency of steps and simplicity of code. For many programs "industrial" optimizations leading to better running times are possible using special tricks and less understandable code.

How to read the book. Readers should first become familiar with the mathematical background presented in Chapter 1, which is needed to understand concepts and constructions described later. Chapters 2-6 present distinct types of finite-state devices and the central constructions related to these concepts. Materials presented go beyond the contents of many textbooks on finite-state automata, also covering monoidal automata, multi-tape automata, transducers and bimachines. A selective reading is possible, e.g. with a focus on read-only automata. While this part of the book can be read independently, we recommend - after a brief introduction to the C(M) language (Chapter 8) - to read Part I and Chapter 9 in parallel. Chapter 9 presents executable C(M) implementations for the constructions described in Chapters 2-6, which means that the behaviour of algorithms on arbitrary input examples can be directly studied on the computer. The final chapters 10-12 are most suitable for readers familiar with the basic theory. They demonstrate how finite-state techniques can be used in practice, solving some central problems related to dictionaries and search. Again for all procedures the full program code is presented.

Acknowledgements We are deeply indebted to our colleagues Tinko Tinchev, Petar Mitankin and Stefan Gerdjikov for the advice, feedback and support they gave to us during the writing of this book. We are especially grateful to Stefan for a detailed critique of a preliminary version. We incorporated practically all of his suggestions, and they greatly improved the presentation.

Contents

I Formal background **1**

1 Formal preliminaries **3**

1.1 Sets, functions and relations 3
1.2 Lifting functions to sets and tuples 7
1.3 Alphabets, words and languages 9
1.4 Word tuples, string relations and string functions 11
1.5 The general monoidal perspective 14
1.6 Summing up 17
1.7 Exercises for Chapter 1 18

2 Monoidal finite-state automata **21**

2.1 Basic concept and examples 21
2.2 Closure properties of monoidal finite-state automata 27
2.3 Monoidal regular languages and monoidal regular expressions . . 30
2.4 Equivalence between monoidal regular languages and monoidal automaton languages 32
2.5 Simplifying the structure of monoidal finite-state automata . . . 33
2.6 Summing up 37
2.7 Exercises for Chapter 2 37

3 Classical finite-state automata and regular languages **39**

3.1 Deterministic finite-state automata 39
3.2 Determinization of classical finite-state automata 41
3.3 Additional closure properties for classical finite-state automata . 43
3.4 Minimal deterministic finite-state automata and the Myhill-Nerode equivalence relation 45
3.5 Minimization of deterministic finite-state automata 51
3.6 Coloured deterministic finite-state automata 55
3.7 Pseudo-determinization and pseudo-minimization of monoidal finite-state automata 58
3.8 Summing up 61

3.9 Exercises for Chapter 3 61

4 Monoidal multi-tape automata and finite-state transducers 65
4.1 Monoidal multi-tape automata 65
4.2 Additional closure properties of monoidal multi-tape automata . 68
4.3 Classical multi-tape automata and letter automata 69
4.4 Monoidal finite-state transducers 72
4.5 Classical finite-state transducers 74
4.6 Deciding functionality of classical finite-state transducers 76
4.7 Summing up 80
4.8 Exercises for Chapter 4 82

5 Deterministic transducers 85
5.1 Deterministic transducers and subsequential transducers 85
5.2 A determinization procedure for functional transducers with the bounded variation property 90
5.3 Deciding the bounded variation property 96
5.4 Minimal subsequential finite-state transducers - Myhill-Nerode relation for subsequential transducers 102
5.5 Minimization of subsequential transducers 109
5.6 Numerical subsequential transducers 117
5.7 Summing up 118
5.8 Bibliographic notes 119
5.9 Exercises for Chapter 5 119

6 Bimachines 121
6.1 Basic definitions 121
6.2 Equivalence of regular string functions and classical bimachines . 127
6.3 Pseudo-minimization of monoidal bimachines 131
6.4 Direct composition of classical bimachines 133
6.5 Summing up 136
6.6 Exercises for Chapter 6 137

II From theory to practice 139

7 The *C(M)* language 143
7.1 Basics and simple examples 143
7.2 Types, terms, and statements in *C(M)* 150

8 *C(M)* implementation of finite-state devices 159
8.1 *C(M)* implementations for automata algorithms 159
8.2 *C(M)* programs for classical finite-state transducers 173

8.3 *C(M)* programs for deterministic transducers 186
8.4 *C(M)* programs for bimachines 195

9 The Aho-Corasick algorithm 207
9.1 Formal construction – first version 207
9.2 Linear computation of the Aho-Corasick automaton 214
9.3 Space-efficient variant – construction of the Aho-Corasick f-automaton 215

10 The minimal deterministic finite-state automaton for a finite language 223
10.1 Formal construction 223
10.2 *C(M)* implementation of the construction - first version 228
10.3 Efficient construction of the minimal dictionary automaton . . . 231
10.4 Adapting the language of minimal dictionary automata 233
10.5 The minimal subsequential transducer for a finite two-sided dictionary 237

11 Constructing finite-state devices for text rewriting 245
11.1 Simple text rewriting based on regular relations 246
11.2 Using deterministic machines for simple text rewriting 247
11.3 Left-most-longest-match text rewriting 252
11.4 Regular relations for left-most-longest match rewriting 254

Part I

Formal background

Chapter 1

Formal preliminaries

The aim of this chapter is twofold. First, we recall a collection of basic mathematical notions that are needed for the discussions of the following chapters. Second, we have a first - still purely mathematical - look at the central topics of the book: languages, relations and functions between strings, as well as important operations on languages, relations and functions. We also introduce monoids, a class of algebraic structures that gives an abstract view on strings, languages, and relations.

1.1 Sets, functions and relations

Sets and *Boolean operations on sets* (i.e. union, intersection, difference, complement) are introduced as usual. The cardinality of a set A is written $|A|$.

As usual, *n-tuples* of elements are written $\langle m_1, \ldots, m_n \rangle$. Tuples represent ordered sequences of elements. The empty tuple, which is unique, is written $\langle\rangle$ or ε. 1-tuples $\langle m \rangle$ coincide with elements m. In what follows, n-tuples $\langle m_1, \ldots, m_n \rangle$ are often written as $\bar{m}$. If $n \geq 2$ and $\bar{m}$ is as above, then $\bar{m}_{\times i}$ is the $(n-1)$-tuple obtained from $\bar{m}$ by deleting the i-th component.

Remark 1.1.1 Let $l \geq 2$, let $n = k_1 + k_2 + \ldots k_l$ $(k_i \geq 1)$ and let $\bar{a}_i = \langle a_{i,1}, a_{i,2}, \ldots, a_{i,k_i} \rangle$ denote a k_i-tuple for $i = 1, \ldots, l$. In this situation we formally distinguish between the l-tuple

$$\langle \bar{a}_1, \ldots, \bar{a}_l \rangle = \langle \langle a_{1,1}, \ldots, a_{1,k_1} \rangle, \langle a_{2,1}, \ldots, a_{2,k_2} \rangle, \ldots, \langle a_{l,1}, \ldots, a_{l,k_l} \rangle \rangle$$

and the n-tuple

$$\langle a_{1,1}, \ldots, a_{1,k_1}, a_{2,1}, \ldots, a_{2,k_2}, \ldots, a_{l,1}, \ldots, a_{l,k_l} \rangle .$$

Let $n \geq 1$, let $M_1, \ldots, M_n$ be sets. The *Cartesian product* of $M_1, \ldots, M_n$ is

$$\prod_{i=1}^{n} M_i := \{ \langle m_1, \ldots, m_n \rangle \mid m_i \in M_i \text{ for } 1 \leq i \leq n \}.$$

An alternative notation is $M_1 \times \ldots \times M_n$. If $M_1 = \ldots = M_n = M$, then $\prod_{i=1}^{n} M_i$ is also written M^n.

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

Definition 1.1.2 Let $M, M_1, \ldots, M_n$ be sets. Any subset $R \subseteq \prod_{i=1}^{n} M_i$ is called an *n-ary relation*. If $M_1 = \ldots = M_n = M$, then R is called an *n-ary relation on* M. The set $\{\langle m, m\rangle \mid m \in M\}$ is called the *identity* on M and denoted Id_M.

If R is an n-ary relation we often write $R(m_1, \ldots, m_n)$ or $R(\bar{m})$ to indicate that $\bar{m} = \langle m_1, \ldots, m_n\rangle \in R$. In the special case of a binary relation R we also use the notation xRy to express that $\langle x, y\rangle \in R$.

As a matter of fact we use the *Boolean operations* (union, intersection, difference, complement) for relations. For binary relations there are other standard operations.

Definition 1.1.3 Let $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$ be binary relations. Then

$$R_1 \circ R_2 := \{\langle a, c\rangle \mid \exists b \in B : R_1(a, b), R_2(b, c)\}$$

is the *composition* of R_1 and R_2.

Note that $R_1 \circ R_2$ is again a binary relation. It is simple to see that composition of relations is an associative operation.

Definition 1.1.4 Let R be a binary relation. Then

$$R^{-1} := \{\langle m, n\rangle \mid \langle n, m\rangle \in R\}$$

is the *inverse relation* of R.

Definition 1.1.5 Let $R \subseteq A \times B$ be a binary relation, let $X \subseteq A$. Then $R(X) := \{b \in B \mid \exists x \in X : R(x, b)\}$ is called the *image of* X *under* R.

Definition 1.1.6 A binary relation $R \subseteq M \times N$ is *infinitely ambiguous* iff there exists an $m \in M$ such that the set $R(\{m\})$ is infinite.

Definition 1.1.7 Let $R \subseteq M \times M$ be a binary relation.

1. R is *reflexive* (w.r.t. M) iff for all $m \in M$ always $R(m, m)$.
2. R is *symmetric* iff for all $m, n \in M$ always $R(m, n)$ implies $R(n, m)$.
3. R is *transitive* iff for all $m, m', m'' \in M$ always $R(m, m')$ and $R(m', m'')$ imply $R(m, m'')$.
4. R is *antisymmetric* iff for all $m, m' \in M$ always $R(m, m')$ and $R(m', m)$ imply $m = m'$.

To check reflexivity it is not sufficient to know R, the set M must be clear from the context. Note that the identity Id_M is both symmetric and antisymmetric. Given a binary relation $R \subseteq M \times M$ we often look for extensions of R to a transitive (or reflexive and transitive) relation on M. This motivates the following definition.

Definition 1.1.8 For a binary relation R on a given set M we inductively define

- $R^0 := Id_M$,
- $R^{i+1} := R^i \circ R$,
- $R^* := \bigcup_{i=0}^{\infty} R^i$,
- $R^+ := \bigcup_{i=1}^{\infty} R^i$.

R^* is called the *reflexive and transitive closure* or the *(relational) Kleene star* of R. Similary R^+ is called the *transitive closure* of R.

Clearly, the (reflexive and) transitive closure is the smallest extension of R to a binary relation on M that is (reflexive and) transitive. For computing the reflexive and transitive closure of a binary relation $R \subseteq M \times M$ the set M must be known. If $n \geq 2$, projections represent another useful type of operations for n-ary relations.

Definition 1.1.9 Let $R \subseteq \prod_{i=1}^{n} M_i$ where $n \geq 2$. The relation

$$R_{\times i} := \{\bar{m}_{\times i} \mid R(\bar{m})\}$$

is called the *projection of R w.r.t. the set of coordinates* $\{1, \ldots, i-1, i+1, \ldots, n\}$. Let $\emptyset \neq \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. Then

$$Proj(\langle i_1, \ldots, i_k\rangle, R) := \{\langle m_{i_1}, \ldots, m_{i_k}\rangle \mid \langle m_1, \ldots, m_n\rangle \in R\}$$

is called the *generalized projection of R w.r.t. the sequence of coordinates* $\langle i_1, \ldots, i_k\rangle$.

Note that in the above situation we have $R_{\times i} = Proj(\langle 1, \ldots, i-1, i+1, \ldots, n\rangle, R)$. If $n = 2$ we have $R^{-1} = Proj(\langle 2, 1\rangle, R)$. In what follows, generalized projections are simply called *projections*.

Example 1.1.10 Let $R = \{\langle a_1, b_1, c_1, d_1\rangle, \langle a_1, b_1, c_2, d_1\rangle, \langle a_2, b_1, c_2, d_2\rangle\}$. Then

$$\begin{aligned} Proj(\langle 3, 2\rangle, R) &= \{\langle c_1, b_1\rangle, \langle c_2, b_1\rangle\} \\ Proj(\langle 4, 1\rangle, R) &= \{\langle d_1, a_1\rangle, \langle d_2, a_2\rangle\}. \end{aligned}$$

Definition 1.1.11 Let $R \subseteq M \times M$. R is an *equivalence relation* on M iff R is reflexive, symmetric, and transitive. Let $R \subseteq M \times M$ be an equivalence relation. For each $m \in M$ the set $[m]_R := \{n \in M \mid R(m, n)\}$ is called the *equivalence class* of m. $M/R := \{[m]_R \mid m \in M\}$ denotes the set of all equivalence classes. The *index* of R is the cardinality $|M/R|$.

Equivalence relations are often written in infix notation, using symbols such as $\sim$ or $\equiv$.

Definition 1.1.12 Let $R_1, R_2 \subseteq M \times M$ be equivalence relations on the set M. R_2 is a *refinement* of R_1 iff $[m]_{R_2} \subseteq [m]_{R_1}$ for all $m \in M$.

Definition 1.1.13 Let $R \subseteq M \times M$. R is a *partial order* on M iff R is reflexive, transitive, and antisymmetric. A partial order $R \subseteq M \times M$ is a *linear order* iff for all $m, n \in M$ always $R(m, n)$ or $R(n, m)$.

Definition 1.1.14 Let M and N be sets. A subset $f \subseteq M \times N$ is called a *partial function* from M to N if $\langle m, n\rangle \in f$ and $\langle m, n'\rangle \in f$ implies $n = n'$. The notation $f : M \to N$ indicates that f is a partial function from M to N. As usual we write $n = f(m)$ if $\langle m, n\rangle \in f$. The element n is called the image of m under f. The *domain* $dom(f)$ and the *codomain* $codom(f)$ of a partial function $f : M \to N$ are respectively defined as

$$\begin{aligned} dom(f) &:= \{m \in M \mid \exists n \in N : n = f(m)\} \\ codom(f) &:= \{n \in N \mid \exists m \in M : n = f(m)\}. \end{aligned}$$

If $f : M \to N$ and $X \subseteq M$, then the *restriction of f to X* is the partial function $f|_X : X \to N$ that maps each $x \in (X \cap dom(f))$ to $f(x)$. The *image of X under f* is

$$f(X) := \{n \in N \mid \exists x \in X : n = f(x)\} = \{f(x) \mid x \in X\}.$$

A *total function* from M to N is a partial function with domain M.

Total functions f are often described in a rule-based way $f : m \mapsto n$, specifying the image n of a "generic" element m. For example $s : \mathbb{N} \to \mathbb{N} : n \mapsto n + 1$ denotes the successor function for natural numbers. Note that in the above situation we have $dom(f) = Proj(1, f)$ and $codom(f) = Proj(2, f)$. In what follows, by a function we always mean a partial function if not mentioned otherwise. As a general convention, when writing an expression $f(m)$ we always mean that f is defined for m.

Definition 1.1.15 Let M, N be sets. A *k-ary function from M to N* is a function $f : M^k \to N$.

If $f : M^k \to N$ is a k-ary function we write $f(m_1, \ldots, m_k)$ for $f(\langle m_1, \ldots, m_k\rangle)$. When using a relation R in a formal construction, concise descriptions can often be achieved when introducing *functions* that are derived from R in a particular way. The following definition describes a general and flexibel way to derive functions from a given relation.

Definition 1.1.16 Let $R \subseteq \prod_{i=1}^{n} M_i$ where $n \geq 2$. Let $I = \langle i_1, \ldots, i_k\rangle$ and $J = \langle j_1, \ldots, j_l\rangle$ be two sequences where $\{i_1, \ldots, i_k\}$ and $\{j_1, \ldots, j_l\}$ are non-empty subsets of the index set $\{1, \ldots, n\}$. Then $Func(I, J, R)$ denotes the function that maps each element $\langle m_{i_1}, \ldots, m_{i_k}\rangle$ of $Proj(I, R)$ to the set

$$\{\langle m_{j_1}, \ldots, m_{j_l}\rangle \in Proj(J, R) \mid \exists \langle m_1, \ldots m_n\rangle \in R\}.$$

Example 1.1.17 Extending Example 1.1.10, let

$$R = \{\langle a_1, b_1, c_1, d_1\rangle, \langle a_1, b_1, c_2, d_1\rangle, \langle a_2, b_1, c_2, d_2\rangle\}$$

as above. Then $Func(\langle 3, 2\rangle, \langle 4, 1\rangle), R)$ is the following mapping:

$$\begin{aligned} \langle c_1, b_1\rangle &\mapsto \{\langle d_1, a_1\rangle\} \\ \langle c_2, b_1\rangle &\mapsto \{\langle d_1, a_1\rangle, \langle d_2, a_2\rangle\} \end{aligned}$$

Figure 1.1: Lifting a function to sets. The partial function f (left-hand side) maps elements to elements. The lifted version (right-hand side) maps sets of elements to an image set, proceeding in an element-wise manner.

Definition 1.1.18 Let $f : A \to B$ and $g : B \to C$ be functions. Then

$$f \circ g := \{\langle a, g(f(a))\rangle \mid a \in dom(f), f(a) \in dom(g)\}$$

is called the (functional) *composition* of f and g. Note that $f \circ g : A \to C$.

As a matter of fact, composition of functions is a special case of the more general composition of relations introduced in Definition 1.1.3, which implies that it is an associative operation.

Definition 1.1.19 Let $f : A \to B$ be a function.

1. f is *injective* iff for all $a, a' \in A : (f(a) = f(a')$ implies $a = a')$.
2. f is *surjective* (w.r.t. B) iff for all $b \in B\ \exists a \in A : b = f(a)$.
3. f is *bijective* iff f is surjective and injective.

Bijective functions $f : A \to A$ are also called *permutations* of the set A. When checking surjectivity of a function $f : A \to B$ the set B must be known.

1.2 Lifting functions to sets and tuples

We now introduce two ways of "lifting" a function that are used later at many places.

Definition 1.2.1 Let $f : M \to N$ be a function. The "set-lifted" version of f is the function $\hat{f} : 2^M \to 2^N$ defined pointwise:

$$\hat{f}(T) = \{f(t) \,|\, t \in T \cap dom(f)\}.$$

The function $\hat{f}$ is a total function, $\hat{f}(T)$ is defined for any $T \subseteq M$.

The set-lifting of functions is illustrated in Figure 1.1. Later we will simply use the symbol f to denote a lifted version if this does not lead to confusion.

Example 1.2.2 Let s denote the successor function $n \mapsto n + 1$ on $\mathbb{N}$. Let $E := \{2n \mid n \in \mathbb{N}\}$ and $O := \{2n + 1 \mid n \in \mathbb{N}\}$ respectively denote the set of even and odd natural numbers. Then $\hat{s}(E) = O$ and $\hat{s}(O) = E \setminus \{0\}$.

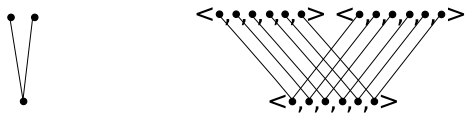

Figure 1.2: Lifting a binary function to a binary function on n-tuples. The binary function f maps two arguments to an image element (left-hand side). The lifted version (right-hand side) maps two n-tuples to an image n-tuple, proceeding in a component-wise manner.

Proposition 1.2.3 *Let $f : M \to N$ be a function, let $T_i \subseteq M$ for every $i \in I$. Then*

$$f(\bigcup_{i \in I} T_i) = \bigcup_{i \in I} f(T_i).$$

Sets as considered for the above lifting technique are unordered collections of elements. A corresponding lifting technique exists for ordered collections.

Definition 1.2.4 Let $f : M^k \to N$ be a k-ary function, let $n \geq 2$. The "tuple-lifted" version of f is the k-ary function $\bar{f} : (M^n)^k \to N^n$ defined component-wise:

$$\begin{array}{rl} & \bar{f}(\langle m_{1,1}, \ldots, m_{1,n}\rangle, \ldots, \langle m_{k,1}, \ldots, m_{k,n}\rangle) \\ = & \langle f(m_{1,1}, \ldots, m_{k,1}), \ldots, f(m_{1,n}, \ldots, m_{k,n})\rangle. \end{array}$$

In general the function $\bar{f}$ is a partial function. This concept is illustrated in Figure 1.2 for the case of a binary function f.

Example 1.2.5

- Let s denote the successor function $n \mapsto n + 1$ on $\mathbb{N}$. Then $\bar{s}(\langle 2, 7, 12\rangle) = \langle 3, 8, 13\rangle$.
- Let $+$ denote the addition of natural numbers. Then $\langle 2, 7, 12\rangle \bar{+} \langle 3, 1, 12\rangle = \langle 5, 8, 24\rangle$.

Remark 1.2.6 Definition 1.2.4 can easily be generalized to the situation where for each of the n components of the tuples a specific k-ary function $f_i : M_i^k \to N_i$ is given. In this case the corresponding definition for the lifted mapping $\bar{f}$ is simply

$$\begin{array}{rl} & \bar{f}(\langle m_{1,1}, \ldots, m_{1,n}\rangle, \ldots, \langle m_{k,1}, \ldots, m_{k,n}\rangle) \\ := & \langle f_1(m_{1,1}, \ldots, m_{k,1}), \ldots, f_n(m_{1,n}, \ldots, m_{k,n})\rangle. \end{array}$$

Here $\bar{f}$ is a k-ary function from the Cartesian product $\Pi_{i=1}^n M_i$ to $\Pi_{i=1}^n N_i$. We also write $f_1 \times \ldots \times f_n$ for the lifted function $\bar{f}$.

Example 1.2.7 Let $f_1 = +$ denote the addition of natural numbers, let $f_2 = \cdot$ denote multiplication of integers. Then $\bar{f}(\langle 2, -3\rangle, \langle 11, 7\rangle) = \langle 13, -21\rangle$.

Once we have seen how to lift functions to sets and tuples it should be clear that lifting can be iterated, obtaining functions that act on sets of tuples, or tuples of sets, etc. We shall come to places where such notions naturally occur.

1.3 Alphabets, words and languages

An *alphabet* Σ is a set of symbols. If not mentioned otherwise, alphabets are assumed to be non-empty and finite.

Definition 1.3.1 Let Σ be an alphabet. A *word* w over Σ is an n-tuple

$$w = \langle a_1, \ldots, a_n \rangle$$

where $n \geq 0$ and $a_i \in \Sigma$ for $i = 1, \ldots, n$. The integer n is called the *length* of w and denoted $|w|$. The empty tuple $\langle\rangle = \varepsilon$, which has length 0, is called the *empty word*. Letters σ of the alphabet Σ are also treated as words of length 1. Σ^* denotes the set of all words over Σ, and $\Sigma^\varepsilon := \Sigma \cup \{\varepsilon\}$. The *concatenation* of two words $u = \langle a_1, \ldots, a_n \rangle$ and $v = \langle b_1, \ldots, b_m \rangle \in \Sigma^*$ is

$$u \cdot v := \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle .$$

Clearly, $u \cdot v \in \Sigma^*$ and $|u \cdot v| = |u| + |v|$. We use the expressions "word" and "string" as synonyms, and "texts", from our perspective, are just strings over a given alphabet. After we have formalized words as tuples we henceforth simply write $w = a_1 \ldots a_n$ instead of $w = \langle a_1, \ldots, a_n \rangle$. The concatenation $u \cdot v$ of two words u and v is often written in the simpler form uv.

Definition 1.3.2 The *reverse function* $\rho : \Sigma^* \to \Sigma^*$ is defined by induction as

$$\begin{aligned} &\rho(\varepsilon) := \varepsilon, \\ &\forall u \in \Sigma^* \ \forall a \in \Sigma : \rho(u \cdot a) := a \cdot \rho(u). \end{aligned}$$

In "stringology", many simple notions are related to words and concatenation.

Definition 1.3.3 Let $t \in \Sigma^*$, assume that t can be represented in the form $t = u \cdot v \cdot w$ for some $u, v, w \in \Sigma^*$. Then $v \in \Sigma^*$ is an *infix* of t. If $u = \varepsilon$, then v is a *prefix* of t. If $w = \varepsilon$, then v is a *suffix* of t. The prefix (suffix) v of t is a *proper* prefix (suffix) of t if $v \neq t$. The notation $v \leq t$ ($v < t$) expresses that v is a (proper) prefix of t.

If $t \in \Sigma^*$ is any word, $Pref(t)$ and $Suf(t)$ respectively denote the set of prefixes and suffixes of t. The corresponding set lifted versions are defined correspondingly: if $M \subseteq \Sigma^*$ is any set of strings, then $Pref(M) := \bigcup_{m \in M} Pref(m)$ and $Suf(M) := \bigcup_{m \in M} Suf(m)$ denote the set of prefixes (suffixes) of strings in M, respectively.

Definition 1.3.4 The expression $u^{-1}t$ denotes the word v if u is a prefix of t and $t = u \cdot v$, otherwise $u^{-1}t$ is undefined.

For any two words $u, v \in \Sigma^*$ the set of *common prefixes* is $C = Pref(u) \cap Pref(v)$. Since C is finite and $<$ is a linear order on C there exists a unique maximal element.

Definition 1.3.5 Let $u, v \in \Sigma^*$. Then $w = u \wedge v$ denotes the *longest common prefix* of u and v.

The following properties follow immediately (cf. Exercise 1.7.6).

Proposition 1.3.6 *Let u, v, $w \in \Sigma^*$. Then*

1. $uv \wedge uw = u(v \wedge w)$,
2. $(uv)^{-1}uw = v^{-1}w$,
3. *if* $c := u \wedge v$, *then* $c^{-1}u \wedge c^{-1}v = \varepsilon$,
4. *if* $u \wedge v = \varepsilon$ *and* $u \neq \varepsilon$, $v \neq \varepsilon$, *then* $\forall x, y \in \Sigma^*$: $ux \wedge vy = \varepsilon$.

An infix v of a string t can have several "occurrences" in t which start at distinct "positions" in t. We formalize these notions.

Definition 1.3.7 A *position* of $t \in \Sigma^*$ is a pair $\langle u, v\rangle$ such that $t = uv$. Given a string $t \in \Sigma^*$ the notation $\langle u, v\rangle_t$ indicates that $\langle u, v\rangle$ is a position of $t = uv$. An *infix occurrence* (of the infix v) in $t \in \Sigma^*$ is a triple $\langle u, v, w\rangle$ such that $t = uvw$. Given a string $t \in \Sigma^*$ the notation $\langle u, v, w\rangle_t$ indicates that $\langle u, v, w\rangle$ is an infix occurrence of $t = uvw$.

Remark 1.3.8 If Σ is an alphabet, the triple $\langle \Sigma^*, \cdot, \varepsilon\rangle$ is an algebraic structure called the *free monoid* for the "set of generators" Σ. This name is related to the fact that for each word $w \in \Sigma^*$ there exists a unique representation of w as a finite (possibly empty) concatenation of elements of Σ. The general notion of a monoid is introduced below.

Definition 1.3.9 Let Σ be an alphabet. A subset $L \subseteq \Sigma^*$ is called a *language* over Σ.

Later the notion of a language will be generalized, introducing "monoidal" languages (cf. Definition 1.5.4). In situations where ambiguities might arise, languages of the form $L \subseteq \Sigma^*$ are called "classical languages".

Since languages are sets of words, we may use the Boolean operations (union, intersection, difference, complement) for languages. Furthermore, the above set-lifting techniques can be used to lift the operations introduced for words to the level of languages. From a notational point we do not introduce new symbols.

Definition 1.3.10 Let L_1, L_2 be classical languages over the alphabet Σ. Then

$$L_1 \cdot L_2 := \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$$

is called the *concatenation* of L_1 and L_2.

As concatenation of words, also concatenation of languages is an associative operation.

Example 1.3.11 Let Σ denote the alphabet of Latin letters with the blank symbol "_", let $L_1 := \{\mathit{Susan}, \mathit{Peter}, \ldots\}$ denote a list of pre-names, let $L_2 := \{\_\}$, let $L_1 := \{\mathit{Brown}, \mathit{Miller}, \ldots\}$ denote a list of family names. L_1, L_2 and L_3 are languages over Σ. The language $L_1 \cdot L_2 \cdot L_3$ contains all names of the form "*Susan_Brown*" obtained by combining the names in L_1 and L_2.

Definition 1.3.12 Let L be a classical language over the alphabet Σ. Then the set lifted version of ρ:

$$\rho(L) := \{\rho(w) \mid w \in L\}$$

is called the *reverse language* of L.

Definition 1.3.13 Let L be a language. We inductively define

1. $L^0 = \{\varepsilon\}$,
2. $L^{k+1} = L^k \cdot L$,
3. $L^* = \bigcup_{k=0}^{\infty} L^k$,
4. $L^+ = \bigcup_{k=1}^{\infty} L^k$.

L^* is called the *Kleene star* of L.

Words, languages and language operations provide the formal background for the theory of (standard, one-tape) finite-state automata, which is covered in Chapter 2. We now turn to notions that are relevant for the theory of n-tape automata, which is treated in Chapter 4.1.

1.4 Word tuples, string relations and string functions

For all notions introduced in the previous section there are natural generalizations when moving from elements to n-tuples. First, following the lifting techniques discussed in Remarks 1.2.4, 1.2.6 and visualized in Figure 1.2 we define concatenation of n-tuples of words.

Definition 1.4.1 The *concatenation of n-tuples of words* (or *n-way concatenation of words*) is defined as

$$\langle u_1, \ldots, u_n \rangle \mathbin{\bar{\cdot}} \langle v_1, \ldots, v_n \rangle := \langle u_1 \cdot v_1, \ldots, u_n \cdot v_n \rangle.$$

Note that $\bar{\varepsilon} := \langle \varepsilon, \ldots, \varepsilon \rangle$ is a unit element in the sense that

$$\langle u_1, \ldots, u_n \rangle \mathbin{\bar{\cdot}} \bar{\varepsilon} = \bar{\varepsilon} \mathbin{\bar{\cdot}} \langle u_1, \ldots, u_n \rangle = \langle u_1, \ldots, u_n \rangle$$

for all tuples $\langle u_1, \ldots, u_n \rangle$. It is sometimes helpful to imagine tuples in an n-way concatenation written in vertical direction. Figure 1.3 illustrates Definition 1.4.1 in this way.

Example 1.4.2 Let $\Sigma = \{a, b\}$. Then $\langle aaa, bb \rangle \mathbin{\bar{\cdot}} \langle ba, a \rangle = \langle aaaba, bba \rangle$.

Similarly as concatenation, also reversal of words naturally generalizes to reversal of n-tuples of words.

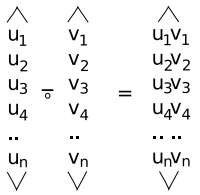

Figure 1.3: Illustration for Definition 1.4.1, n-way concatenation of words, tuples written in vertical direction.

Definition 1.4.3 The *reversal of n-tuples of words* (or *n-way reversal*) is defined as

$$\bar{\rho}(\langle u_1, \ldots, u_n \rangle) := \langle \rho(u_1), \ldots, \rho(u_n) \rangle.$$

Example 1.4.4 Let $\Sigma = \{a, b\}$. Then $\bar{\rho}(\langle aaab, bba \rangle = \langle baaa, abb \rangle$.

Remark 1.4.5 In Remark 1.3.8 we introduced the free monoid for a set of generators (alphabet) Σ. Let $n \geq 2$, let Σ_i be an alphabet for $1 \leq i \leq n$. The set $\prod_{i=1}^{n} \Sigma_i^*$ of all n-tuples of words, together with n-ary concatenation and $\bar{\varepsilon}$ represents an algebraic structure called the *Cartesian product of the free monoids* $\langle \Sigma_i^*, \cdot, \varepsilon \rangle$. Cartesian products of free monoids provide another important example of the general concept of a monoid to be introduced below.

After having lifted words to word tuples, we now generalize the notion of a language to the n-ary case.

Definition 1.4.6 Let $n \geq 1$. For each $i = 1, \ldots, n$, let Σ_i be an alphabet. Then each set $R \subseteq \prod_{i=1}^{n} \Sigma_i^*$ is called an *n-ary string relation*.

Example 1.4.7 Let $\Sigma_1 := \{a\}$, $\Sigma_2 := \{b\}$. Some examples of binary string relations are:

$$\begin{aligned} R_1 &:= \{\langle a, b \rangle\}, \\ R_2 &:= \{\langle a^n, b^n \rangle \mid n \in \mathbb{N}\}, \\ R_3 &:= \{\langle \varepsilon, b^n \rangle \mid n \in \mathbb{N}\}. \end{aligned}$$

Since string relations are sets of n-tuples of words the Boolean operations (union, intersection, difference, complement) can be used for n-ary string relations. Lifting concatenation of words to sets of tuples (or n-way concatenation of word tuples to sets) we obtain concatenation of string relations.

Definition 1.4.8 Let R_1, R_2 be n-ary string relations. The *concatenation* of R_1 and R_2 is

$$R_1 \bar{\cdot} R_2 := \{\bar{u} \bar{\cdot} \bar{v} \mid \bar{u} \in R_1, \bar{v} \in R_2\}.$$

Note that $R_1 \bar{\cdot} R_2$ is again an n-ary string relation. More precisely the operation $\bar{\cdot}$ is also called *set-lifted n-way concatenation*. It is important to note that set-lifted n-way concatenation of relations $R_1 \bar{\cdot} R_2$ is distinct from the usual relational composition $R_1 \circ R_2$ of binary relations R_1 and R_2 as defined in Definition 1.1.3.

Example 1.4.9 As above, let

$$\begin{aligned} R_1 &:= \{\langle a, b\rangle\}, \\ R_2 &:= \{\langle a^n, b^n\rangle \mid n \in \mathbb{N}\}, \\ R_3 &:= \{\langle \varepsilon, b^n\rangle \mid n \in \mathbb{N}\}. \end{aligned}$$

Then

$$\begin{aligned} R_1 \mathbin{\bar{\cdot}} R_2 &= \{\langle aa^n, bb^n\rangle \mid n \in \mathbb{N}\} = \{\langle a^n, b^n\rangle \mid 0 \neq n \in \mathbb{N}\}, \\ R_2 \mathbin{\bar{\cdot}} R_3 &= \{\langle a^n, b^m\rangle \mid m, n \in \mathbb{N}, m \geq n\}, \\ R_1 \mathbin{\bar{\cdot}} R_3 &= \{\langle a, b^n\rangle \mid 0 \neq n \in \mathbb{N}\}. \end{aligned}$$

Similarly as concatenation, also reversal of words generalizes to reversal of sets if n-tuples of words.

Definition 1.4.10 Let R be an n-ary string relation. The *n-way reverse relation* of R is

$$\bar{\rho}(R) := \{\bar{\rho}(\bar{u}) \mid \bar{u} \in R\}.$$

Example 1.4.11 Let $R := \{\langle abc, aabbcc\rangle, \langle bcd, bbccdd\rangle, \langle ad, af\rangle\}$. Then $\bar{\rho}(R) = \{\langle cba, ccbbaa\rangle, \langle dcb, ddccbb\rangle, \langle da, fa\rangle\}$.

Definition 1.4.12 The *(concatenation) Kleene star* for an n-ary string relation R is defined as

- $R^0 = \{\bar{\varepsilon}\}$,
- $R^{k+1} = R^k \mathbin{\bar{\cdot}} R$,
- $R^* = \bigcup_{k=0}^{\infty} R^k$.

We also define $R^+ = \bigcup_{k=1}^{\infty} R^k$.

At this point, the reader will notice a notational clash. In fact, the symbols R^i, R^*, and R^+ have been already used with another meaning in Definition 1.1.8 when introducing the *relational* Kleene Star. Unfortunately, these symbols are well established in both contexts, hence we decided not to introduce special symbols in one case. If in a special context both interpretations are possible we make the intended meaning of the symbols explicit.

Example 1.4.13 As above, let

$$\begin{aligned} R_1 &:= \{\langle a, b\rangle\}, \\ R_2 &:= \{\langle a^n, b^n\rangle \mid n \in \mathbb{N}\}, \end{aligned}$$

and $k \in \mathbb{N}$. Then $R_1^k = \langle a^k, b^k\rangle$ and $R_1^* = R_2$.

Remark 1.4.14 It is useful to imagine a binary (n-ary) string relation as a collection of (multi-ary) string translation rules. Following this picture, an entry $\langle u, v\rangle$ of a binary string relation R means that string u can be translated to v. We may depict such a rule in the form $\rightarrow_v^u$. If $\rightarrow_{v_1}^{u_1}$ and $\rightarrow_{v_2}^{u_2}$ are two string

translation rules it is natural to combine these rules into the joint translation $\rightarrow_{v_1 v_2}^{u_1 u_2}$. This is just 2-way concatenation. Using this notation, the concatenation of two binary string relations R_1 and R_2 can be described as

$$R_1 \stackrel{\cdot}{\cdot} R_2 := \{\rightarrow_{v_1 v_2}^{u_1 u_2} | \rightarrow_{v_1}^{u_1} \in R_1, \rightarrow_{v_2}^{u_2} \in R_2\}.$$

Similarly R^* describes the set of all rules $\rightarrow_{v_1 \ldots v_n}^{u_1 \ldots u_n}$ where $n \geq 0$ and $\rightarrow_{v_i}^{u_i} \in R$ for $i = 1, \ldots n$.

Definition 1.4.15 A binary string relation that is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ is called a *string function*.

1.5 The general monoidal perspective

In Remarks 1.3.8 and 1.4.5 we introduced algebraic structures that describe strings and n-tuples of strings. These two structures are special cases of the general concept of a monoid. Monoids offer a common abstract background for formal languages, string relations and multi-dimensional generalizations. Several constructions for automata and n-tape automata covered later in the book generalize to the situation where an arbitrary monoid is used. In order to avoid a reduplication of constructions and proofs we introduce the general concept of a monoid, and develop a kind of rudimentary "formal language theory" for general monoids.

Monoids

Definition 1.5.1 A *monoid* $\mathcal{M}$ is a triple $\langle M, \circ, e \rangle$ where

- M is a non-empty set, the set of monoid elements,
- $\circ : M \times M \rightarrow M$ is the monoid operation (we will use infix notation),
- $e \in M$ is the monoid unit element,

and the following conditions hold:

- $\forall a, b, c \in M : a \circ (b \circ c) = (a \circ b) \circ c$ (associativity of "$\circ$"),
- $\forall a \in M : a \circ e = e \circ a = a$ (e is a unit element).

Later we often use the set M to denote the monoid $\mathcal{M} = \langle M, \circ, e \rangle$. As a matter of fact, the free monoid for a set of generators (alphabet) Σ (cf. Remark 1.3.8) and the Cartesian product of free monoids (cf. Remark 1.4.5) are special monoids. We list some further examples.

Example 1.5.2 Let M be a set.

1. The set of natural numbers $\mathbb{N}$ with addition $+$ as operation and 0 as unit element is a monoid.
2. The set of all binary relations $R \subseteq M \times M$ with composition $\circ$ as monoid operation and the identity function Id_M as unit element is a monoid $Rel(M)$.

3. The power set 2^M with union $\cup$ as operation and $\emptyset$ as unit is a monoid.
4. The set of all partial or total functions $f : M \to M$ defines a submonoid of $Rel(M)$ respectively written $pFun(M)$ or $Fun(M)$.
5. The set of all bijections of M defines a submonoid of $Rel(M)$ called the permutation group of M.

The following definition shows how monoids are often obtained from simpler monoids using lifting techniques.

Definition 1.5.3 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid, let

$$\widehat{\circ} : 2^M \times 2^M : \langle A, B\rangle \mapsto \{a \circ b \mid a \in A, b \in B\}.$$

Then $\widehat{\mathcal{M}} = \langle 2^M, \widehat{\circ}, \{e\}\rangle$ is called the *set-lifted version of the monoid* $\mathcal{M}$.

Here "$\widehat{\circ}$" can be considered as a lifted version of "$\circ$" where we proceed from pairs of monoid elements to pairs of monoid sets. The set-lifted version of a monoid is again a monoid. Later we often use the same symbol for the basic monoid operation and for the set-lifted version.

Monoidal languages and language operations

Definition 1.5.4 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. A *monoidal language over* $\mathcal{M}$ is a subset of M.

The notion of a monoidal language gives a joint abstract view on languages (Definition 1.3.9) and string relations (Definition 1.4.6). Concatenation of languages and (set-lifted n-way) concatenation of string relations are special instances of the following concept.

Definition 1.5.5 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. For $T_1, T_2 \subseteq M$ the set

$$T_1 \circ T_2 := \{t_1 \circ t_2 \,|\, t_1 \in T_1, t_2 \in T_2\}.$$

is called the *monoidal product* of the monoidal languages T_1, T_2.

In a similar way, the Kleene star of a language (Definition 1.3.13) and the concatenation Kleene star of a string relation (Definition 1.4.12) are special instances of the following general monoidal concept.

Definition 1.5.6 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid and $T \subseteq M$. We inductively define

1. $T^0 = \{e\}$,
2. $T^{k+1} = T^k \circ T$,
3. $T^* = \bigcup_{k=0}^{\infty} T^k$,
4. $T^+ = \bigcup_{k=1}^{\infty} T^k$.

We call T^* the *iteration* or (monoidal) *Kleene star* of T.

Definition 1.5.7 A subset $T \subseteq M$ of a monoid $\mathcal{M} = \langle M, \circ, e\rangle$ is a *submonoid* of $\mathcal{M}$ iff $e \in T$ and $T^2 \subseteq T$.

A submonoid of a monoid is again a monoid.

Proposition 1.5.8 *For any subset $T \subseteq M$ of a monoid $\mathcal{M}$, the set T^* is the smallest submonoid of $\mathcal{M}$ containing T.*

In Algebra, T^* is often called the *submonoid generated by the set T*.

Monoid homomorphisms

Definition 1.5.9 Let $\mathcal{M}_1 = \langle M_1, \circ, e_1\rangle$ and $\mathcal{M}_2 = \langle M_2, \bullet, e_2\rangle$ be monoids. A total function $h : M_1 \to M_2$ is a *monoid homomorphism* iff the following conditions hold:

- $h(e_1) = e_2$,
- $\forall a, b \in M_1 : h(a \circ b) = h(a) \bullet h(b)$.

A *monoid isomorphism* is a bijective monoid homomorphism.

Proposition 1.5.10 *The composition of two monoid homomorphisms is again a monoid homomorphism.*

Example 1.5.11 Let $\mathcal{M}_1$ denote the set of real numbers with addition as operation and 0 as unit element. Let $\mathcal{M}_2$ denote the set of strictly positive real numbers with multiplication as operation and 1 as unit element. Then $exp : r \mapsto e^r$ is a monoid isomomorphism.

Proposition 1.5.12 *Let $\mathcal{M}_1 = \langle M_1, \circ, e_1\rangle$ and $\mathcal{M}_2 = \langle M_2, \bullet, e_2\rangle$ be monoids, let $h : M_1 \to M_2$ be a monoid homomorphism.*

1. *For all $T_1, T_2 \subseteq M_1$ we have $h(T_1 \cup T_2) = h(T_1) \cup h(T_2)$.*
2. *For all $T_1, T_2 \subseteq M_1$ we have $h(T_1 \circ T_2) = h(T_1) \bullet h(T_2)$.*
3. *For all $T \subseteq M_1$ we have $h(T^*) = h(T)^*$.*

Proof. The proof of Points 1 and 2 is straightforward. Let $T_1 = T \subseteq M_1$, let $T_2 := h(T)$. Since $h(e_1) = e_2$ we have $h(T_1^0) = h(\{e_1\}) = \{e_2\} = T_2^0$. Using Point 2 and a simple induction we see that $h(T_1^k) = T_2^k$ for all $k \geq 0$. Hence $h(T_1^*) = h(\bigcup_{k=0}^{\infty} T_1^k) = \bigcup_{k=0}^{\infty} h(T_1^k) = \bigcup_{k=0}^{\infty} T_2^k = T_2^* = h(T)^*$. □

Using Point 2 of the above proposition it is easy to show that the lifted version $\hat{h}$ of a monoid homomorphism $h : \mathcal{M}_1 \to \mathcal{M}_2$ represents a monoid homomorphism between the lifted versions of the monoids $\mathcal{M}_1$ and $\mathcal{M}_2$.

Remark 1.5.13 Let $h : \mathcal{M}_1 \to \mathcal{M}_2$ be a monoid homomorphism, let $T \subseteq M_1$. Then $h(T)$ is called the *homomorphic image* of the monoidal language T.

As a matter of fact, $h(T) \subseteq M_2$ is a monoidal language. An important property of the free monoid is captured in the following proposition.

Proposition 1.5.14 *Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid, let Σ be an alphabet and $f : \Sigma \to M$ be a total function. Then the* natural extension h_f *of f over Σ^*, inductively defined as*

1. $h_f(\varepsilon) = e$
2. $h_f(\sigma \cdot a) = h_f(\sigma) \circ f(a)$, *where* $\sigma \in \Sigma^*, a \in \Sigma$,

is a homomorphism between the monoids Σ^* and $\mathcal{M}$ and the unique homomorphism extending f.

Cartesian products of monoids

Following Remark 1.2.6 we obtain a (tuple-) lifted monoid operation by combining the monoid operations of n monoids in a canonical way. Since we will meet this situation later when discussing n-tape automata we have a closer look at this construction.

Definition 1.5.15 Let $n \geq 1$, for $1 \leq i \leq n$ let $\mathcal{M}_i = \langle M_i, \circ_i, e_i\rangle$ be a monoid. Let $\bar{e} := \langle e_1, \ldots, e_n\rangle$ and let $\bar{\circ} : (\prod_{i=1}^n M_i) \times (\prod_{i=1}^n M_i) \to \prod_{i=1}^n M_i$ denote the function

$$\langle u_1, \ldots, u_n\rangle \,\bar{\circ}\, \langle v_1, \ldots, v_n\rangle := \langle u_1 \circ_1 v_1, \ldots, u_n \circ_n v_n\rangle .$$

Then the triple $\prod_{i=1}^n \mathcal{M}_i := \langle \prod_{i=1}^n M_i, \bar{\circ}, \bar{e}\rangle$ is called the *Cartesian product* of the monoids $\mathcal{M}_i$.

As a matter of fact, the Cartesian product of monoids is again a monoid. In the special situation where all component monoids $\mathcal{M}_i$ represent the same monoid $\mathcal{M}$ the Cartesian product is written $\mathcal{M}^n$.

Remark 1.5.16 Let $\mathcal{M} := \prod_{i=1}^n \mathcal{M}_i$ be a Cartesian product of monoids where $n > 1$, let $1 \leq i \leq n$. Let $\mathcal{M}_{\times i}$ denote the projection of $\mathcal{M}$ with respect to the set of coordinates $1, \ldots, i-1, i+2, \ldots, n$ as introduced in Def. 1.1.9. The mapping $p_i : \mathcal{M} \to \mathcal{M}_{\times i} : \bar{m} \mapsto \bar{m}_{\times i}$ is called the *i-th projection mapping*.

It is simple to see that the i-th projection mapping is a monoid homomorphism.

Remark 1.5.17 Let $n \geq 2$, for $i = 1, \ldots, n$ let $\mathcal{M}_i$ be a free monoid $\langle \Sigma_i^*, \cdot, \varepsilon\rangle$. Let $\prod_{i=1}^n \mathcal{M}_i$ be the Cartesian product of the monoids $\mathcal{M}_i$ as introduced in Definition 1.5.15. Then $\prod_{i=1}^n \mathcal{M}_i$ is *not* a free monoid. The elements of the form $\langle \varepsilon, \ldots, \varepsilon, \sigma_i, \varepsilon, \ldots, \varepsilon\rangle$ (where $\sigma_i \in \Sigma_i$ occupies the i-th position) generate the Cartesian product monoid, however, monoid elements $\langle u_1, \ldots, u_n\rangle$ can be represented in multiple ways as a product of such generators.

1.6 Summing up

Table 1.1 summarizes central operations for languages and n-ary string relations. For the sake of completeness we mention *functional composition* as an

	Mon. lang.	Languages	String relations
Union	$T_1 \cup T_2$	$L_1 \cup L_2$	$R_1 \cup R_2$
Intersection	$T_1 \cap T_2$	$L_1 \cap L_2$	$R_1 \cap R_2$
Difference	$T_1 \setminus T_2$	$L_1 \setminus L_2$	$R_1 \setminus R_2$
Complement	$M \setminus T$	$\Sigma^* \setminus L$	$(\Sigma^* \times \ldots \times \Sigma^*) \setminus R$
Concatenation	$T_1 \cdot T_2$	$L_1 \cdot L_2$	$R_1 \cdot R_2$
(Concatenation) Kleene-Star	T_1^*	L^*	R^*
Homomorphism	$h(T)$	$h(L)$	$h(R)$
Projection			$R_{\times i}$ (Def. 1.1.9)
Generalized projection			$Proj(\langle i_1, \ldots, i_k \rangle, R)$ (Def. 1.1.9)
Functionalization			$Func(I, J, R)$ (Def. 1.1.16)
Composition			$R_1 \circ R_2$ (Def. 1.1.3)
Relational Kleene-Star			R^* (Def. 1.1.8)
Reversal	—	$\rho(L)$	$\bar{\rho}(R)$

Table 1.1: Important operations for monoidal languages, classical languages and n-ary string relations. In Column 2, M represents the underlying set of the monoid. Union, intersection, difference, complement, concatenation, (concatenation) Kleene star and homomorphic images for languages and string relations are just special cases of the corresponding operations for monoidal languages. For languages and string relations, reversal represents an additional operation. Furthermore, Cartesian product, composition, relational Kleene star, projection and functionalization are additional operations for string relations.

important operation for string functions. The complete family of these operations represents a kind of calculus with many interesting applications in text processing. In the following section we show how to realize these operations for specific classes of input languages or relations in a procedural way.

1.7 Exercises for Chapter 1

Exercise 1.7.1 Let $R_1 \subseteq A \times B$, $R_2 \subseteq B \times C$ be binary relations. Characterize situations where

1. $|R_1 \circ R_2| = |R_1| \cdot |R_2|$,
2. $|R_1 \circ R_2| = 1$,
3. $|R_1 \circ R_2| = 0$.

Exercise 1.7.2 Let $R \subseteq M \times M$ be a relation such that $n = |R|$ is finite. Show that $|R^+| \leq n^2$. Is $|R^*|$ always finite?

Exercise 1.7.3 Let A, B, C be sets, let $R = A \times B \times C$. What is $Func(\{1\}, \{2, 3\}, R)$? What is $Func(\{1, 2\}, \{3\}, R)$? Let $a \in A, b \in B$. What is $Func(\{1\}, \{2, 3\}, R)(a)$, what is $Func(\{1, 2\}, \{3\}, R)(\langle a, b \rangle)$?

Exercise 1.7.4 Let $f : A \to B$ and $g : B \to C$ be (partial) functions. Prove the following properties:

1. If both f and g are injective, then $f \circ g$ is injective.
2. If both f and g are surjective, then $f \circ g$ is surjective.
3. If $f \circ g$ is bijective, then f is injective and g is surjective.

Exercise 1.7.5 Let $\bar{+}$ denote the binary addition of natural numbers lifted to 3-tuples as defined in Definition 1.2.4. Let O and E respectively denote the set of odd and even natural numbers. What is $\bar{+}(\langle\langle o_1, o_2, o_3\rangle, \langle e_1, e_2, e_3\rangle\rangle \mid o_1, o_2, o_3 \in O, e_1, e_2, e_3 \in E)$?

Exercise 1.7.6 Prove Proposition 1.3.6.

Exercise 1.7.7 Let Σ be an alphabet, let $L_1, L_2 \subseteq \Sigma^*$ be finite languages. Give sufficient conditions that imply $|L_1 \cdot L_2| = |L_1| \cdot |L_2|$.

Exercise 1.7.8 Let $\bar{\cdot}$ denote set-lifted 2-way concatenation as introduced in Definition 1.4.8. Let

$$\begin{aligned} R_1 &:= \{\langle a, b^n\rangle \mid n \in \mathbb{N}\}, \\ R_2 &:= \{\langle a^n, b\rangle \mid n \in \mathbb{N}\}. \end{aligned}$$

Compute $R_1 \bar{\cdot} R_2$.

Exercise 1.7.9 Let a, b be letters of an alphabet, let $R := \{\langle a, b\rangle, \langle \varepsilon, b\rangle\}$. Let $R^2, R^3, \ldots$ and R^+ be defined as in Definition 1.4.12. Compute $R^2, R^3, \ldots$ and R^+.

Exercise 1.7.10 Give an example for an infinite monoid $\mathcal{M} = \langle M, \circ, e\rangle$ where T^* is finite for each finite subset $T \subseteq M$.

Chapter 2

Monoidal finite-state automata

In the previous chapter we introduced a large set of operations on strings, languages, and string relations. We now look at the procedural side. As a starting point we study finite-state automata, which represent the simplest devices for recognizing languages. The theory of finite-state automata has been described in numerous textbooks both from a computational (e.g. [Hopcroft et al., 2006, Kozen, 1997, Lewis and Papadimitriou, 1998]) and an algebraic point of view (e.g., [Eilenberg, 1974, Eilenberg, 1976, Sakarovitch, 2009]). The notes below offer a brief introduction. Here we immediately look at the more general concept of a *monoidal* finite-state automaton, and the focus of this chapter are *general* constructions and results for finite-state automata over *arbitrary monoids* and monoidal languages. Similar generalized perspectives are found in [Eilenberg, 1974]. Refined pictures for the special (and more standard) cases where we only consider free monoids or Cartesian products of monoids will be given later.

2.1 Basic concept and examples

We introduce the central concept of this chapter.

Definition 2.1.1 A *monoidal finite-state automaton* (MSA) is a tuple of the form $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ where

- $\mathcal{M} = \langle M, \circ, e \rangle$ is a monoid,
- Q is a finite *set of states*,
- $I \subseteq Q$ is the set of *initial states*,
- $F \subseteq Q$ is the set of *final states*, and
- $\Delta \subseteq Q \times M \times Q$ is a finite set called the *transition relation*.

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

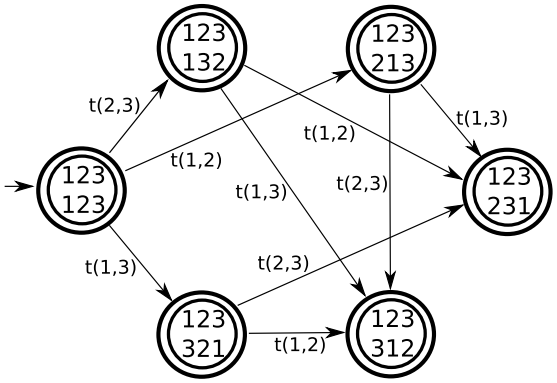

Figure 2.1: A monoidal finite-state automaton over the monoid of all permutations of the set $\{1, 2, 3\}$.

Triples $\langle p, m, q\rangle \in \Delta$ are called *transitions*. The transition $\langle p, m, q\rangle$ *begins* at p, *ends* at q and has the *label* m.

Transitions $\langle p, m, q\rangle$ are also denoted in the form $p \to^m q$.

Example 2.1.2 Figure 2.1 shows a monoidal finite-state automaton with six states over the monoid of all permutations of the set $\{1, 2, 3\}$ (cf. Part 5 of Example 1.5.2). The single start state is marked by an incoming unlabeled arrow. Each state is final, which is indicated by a double contour. The monoid operation is functional composition. The states correspond to the six permutations of $\{1, 2, 3\}$. A transition label of the form $t(i, j)$ denotes the transposition of i and j. Composing transition labels in this way (s.b.), from the start state we reach the state (permutation) that represents the composition of the transpositions. The figure shows that each permutation can be represented as a sequence of transpositions, a well-known fact also valid for larger finite sets.

Definition 2.1.3 *Classical finite-state automata* are monoidal finite-state automata where the underlying monoid is the free monoid over a finite alphabet Σ and the transition relation labels are in $\Sigma^\varepsilon = \Sigma \cup \{\varepsilon\}$.

It should be noted that alternatively classical finite-state automata could be defined in a more liberal way, with transition labels in Σ^*. We shall see below (cf. Remark 2.5.8) that this would not modify the computational power. For some constructions it is advantageous to immediately have all transition labels in $\Sigma \cup \{\varepsilon\}$.

Example 2.1.4 Figure 2.2 shows a classical finite-state automaton as a graph. The automaton has three states 1, 2, and 3. The single start state is 1. The single final state 3 is marked by a double contour. The input alphabet consists of the letters a and b.

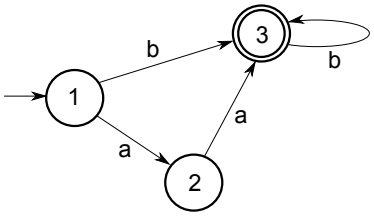

Figure 2.2: A classical automaton with three states.

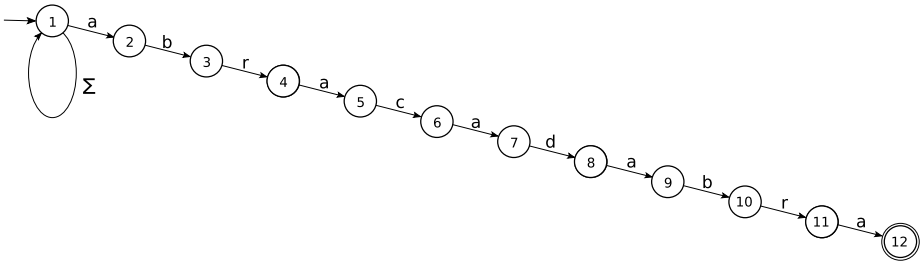

Figure 2.3: A classical automaton for searching occurrences of the pattern *abracadabra* in a text.

Example 2.1.5 Figure 2.3 shows a classical finite-state automaton with 12 states. This automaton can be used for finding occurrences of the pattern *abracadabra* in a text. When reading a text $t \in \Sigma^*$, the final state 12 is reached (s.b.) always after having read an occurrence of *abracadabra* in t.

Classical finite-state automata with monoid Σ^* are also denoted in the form $\langle \Sigma, Q, I, F, \Delta \rangle$.

Example 2.1.6 Let $\mathcal{M} = \mathcal{M}_1^2$ be the Cartesian product of two copies of the free monoid $\mathcal{M}_1 = \{a, b\}^*$, let $Q = I = F = \{1\}$, let $\Delta = \{\langle 1, \langle a, b \rangle, 1 \rangle\}$. Then $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ is a monoidal finite-state automaton. The automaton is shown in Figure 2.4. Using terminology introduced below (cf. Definition 4.1.1), $\mathcal{A}$ is a two-tape automaton.

The language of a monoidal finite-state automaton. As we mentioned above, monoidal finite-state automata are used for recognizing elements belonging to a given monoidal language. The language of a monoidal finite-state automaton can be defined in distinct ways. One way is based on the notions of paths and path labels.

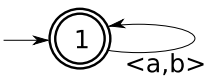

Figure 2.4: A monoidal automaton with just one state.

Definition 2.1.7 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal finite-state automaton. A *proper path* in $\mathcal{A}$ is a finite sequence of $k > 0$ transitions

$$\pi = \langle q_0, a_1, q_1 \rangle \langle q_1, a_2, q_2 \rangle \dots \langle q_{k-1}, a_k, q_k \rangle$$

where $\langle q_{i-1}, a_i, q_i \rangle \in \Delta$ for $i = 1 \dots k$. The number k is called the *length* of π, we say that π starts in q_0 and ends in q_k. States $q_0, \dots, q_k$ are *the states on the path* π. The monoid element $w = a_1 \circ \dots \circ a_k$ is called the *label* of π. We may denote the path π as

$$\pi = q_0 \to^{a_1} q_1 \dots \to^{a_k} q_k.$$

The *null path* of $q \in Q$ is 0_q starting and ending in q with label e. A *successful path* is a path starting in an initial state and ending in a final state. A *loop path* is a proper path that starts and ends in the same state.

Example 2.1.8 Consider the monoidal finite-state automaton introduced in Example 2.1.2.

$$\begin{matrix} 123 \\ 123 \end{matrix} \to^{t(23)} \begin{matrix} 123 \\ 132 \end{matrix} \to^{t(12)} \begin{matrix} 123 \\ 231 \end{matrix}$$

is a path of length 2 leading from the start state $\begin{smallmatrix} 123 \\ 123 \end{smallmatrix}$ to the final state $\begin{smallmatrix} 123 \\ 231 \end{smallmatrix}$. The label is the product of the transpositions $t(23)$ and $t(12)$, which is the permutation $1 \mapsto 2$, $2 \mapsto 3$, $3 \mapsto 1$.

Definition 2.1.9 Let $\mathcal{A}$ be as above. A state $q \in Q$ is called *reachable* from a start state iff there exists a path starting in an initial state and ending in q. There exists a final state *reachable* from a state q iff there exists a path starting in q and ending in a final state.

Note that q satisfies both conditions iff q is on a successful path of $\mathcal{A}$.

Definition 2.1.10 Let $\mathcal{A}$ be as above. Then the set of all labels of successful paths of $\mathcal{A}$ is called the *monoidal language accepted* (or *recognized*) by $\mathcal{A}$ and is denoted $L(\mathcal{A})$.

In general, an element of $L(\mathcal{A})$ can occur as the label of several successful paths. Specialized notions of automata where each element of $L(\mathcal{A})$ represents the label of a unique path are considered later.

Example 2.1.11 Consider the classical automaton shown in Figure 2.2. Here

$$\pi = 1 \to^a 2 \to^a 3 \to^b 3 \to^b 3$$

is a successful path of length 4 with label *aabb*, and we have

$$L(\mathcal{A}) = \{aab^n \mid n \in \mathbb{N}\} \cup \{bb^n \mid n \in \mathbb{N}\}.$$

Second, consider the monoidal automaton shown in Figure 2.3. Here

$$\pi = 1 \to^a 1 \to^a 2 \to^b 3 \to^r 4 \to^a 5 \dots \to^r 11 \to^a 12$$

is a successful path of length 12 with label *aabracadabra*. $L(\mathcal{A})$ is the set of all words over the alphabet Σ ending with suffix *abracadabra*.

Third, consider the monoidal automaton shown in Figure 2.4. Here

$$\pi = 1 \to^{\langle a,b\rangle} 1 \to^{\langle a,b\rangle} 1 \to^{\langle a,b\rangle} 1$$

is a successful path of length 3. Recall that $\langle v_1, v_2\rangle \mathbin{\dot{-}} \langle w_1, w_2\rangle := \langle v_1 w_1, v_2 w_2\rangle$ represents the monoid operation of the Cartesian product. This shows that the path has label $\langle aaa, bbb\rangle$. We have

$$L(\mathcal{A}) = \{\langle a^n, b^n\rangle \mid n \in \mathbb{N}\}.$$

Definition 2.1.12 Two monoidal finite-state automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *equivalent* if $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

Definition 2.1.13 A monoidal language over M is called a *monoidal automaton language* iff it is recognized by some monoidal finite-state automaton. A classical language is called a *classical automaton language* iff it is recognized by a classical finite-state automaton.

The following notion offers a second way to describe the language of a monoidal finite-state automaton.

Definition 2.1.14 Let $\mathcal{A}$ be a monoidal finite-state automata. The *generalized transition relation* Δ^* is defined as the smallest subset of $Q \times M \times Q$ with the following closure properties:

- for all $q \in Q$ we have $\langle q, e, q\rangle \in \Delta^*$.
- For all $q_1, q_2, q_3 \in Q$ and $w, a \in M$: if $\langle q_1, w, q_2\rangle \in \Delta^*$ and $\langle q_2, a, q_3\rangle \in \Delta$, then also $\langle q_1, w \circ a, q_3\rangle \in \Delta^*$.

Triples $\langle p, u, q\rangle \in \Delta^*$ are called *generalized transitions*. The generalized transition $\langle p, u, q\rangle$ *begins* at p, *ends* at q and has the *label* u.

Clearly, generalized transitions are simplified descriptions of (proper or null) paths, representing the beginning, label, and ending of a path while abstracting from intermediate states. Hence the language $L(\mathcal{A})$ can be described as

$$L(\mathcal{A}) = \{m \in M \mid \exists i \in I, f \in F : \langle i, m, f\rangle \in \Delta^*\}.$$

Yet another way of describing $L(\mathcal{A})$ is based on the notion of the language of a (set of) state(s).

Definition 2.1.15 Let $\mathcal{A}$ be as above. For $q \in Q$ the set

$$L_{\mathcal{A}}(q) := \{w \in M \mid \exists f \in F : \langle q, w, f\rangle \in \Delta^*\}$$

is called the *language of state* q *in* $\mathcal{A}$. For $S \subseteq Q$ we define the set lifted version

$$L_{\mathcal{A}}(S) := \bigcup_{q \in S} L_{\mathcal{A}}(q).$$

It follows that for each state q we have $L_{\mathcal{A}}(q) \neq \emptyset$ iff there exists a final state reachable from q, and the language of the automaton can be described as

$$L(\mathcal{A}) = L_{\mathcal{A}}(I).$$

Example 2.1.16 For the classical automaton $\mathcal{A}$ shown in Figure 2.2 we have

$$\begin{aligned} L_{\mathcal{A}}(3) &= \{b^n \mid n \in \mathbb{N}\} \\ L_{\mathcal{A}}(2) &= \{ab^n \mid n \in \mathbb{N}\} \\ L_{\mathcal{A}}(1) &= \{aab^n \mid n \in \mathbb{N}\} \cup \{bb^n \mid n \in \mathbb{N}\} \end{aligned}$$

For the monoidal automaton $\mathcal{A}$ shown in Figure 2.4 we have

$$L_{\mathcal{A}}(1) = \{\langle a^n, b^b \rangle \mid n \in \mathbb{N}\}.$$

The following result, which gives an inductive definition for the language of a state, follows immediately from Definition 2.1.15.

Proposition 2.1.17 *Let $\mathcal{A}$ be as above, let $p \in Q$. Then*

$$L_{\mathcal{A}}(p) = E(p) \quad \cup \bigcup_{\langle p,w,q\rangle \in \Delta} w \cdot L_{\mathcal{A}}(q)$$

where $E(p) := \{\varepsilon\}$ if $p \in F$ and $E(p) := \emptyset$ otherwise.

Definition 2.1.18 Two states $q_1, q_2 \in Q$ are called *equivalent* iff $L_{\mathcal{A}}(q_1) = L_{\mathcal{A}}(q_2)$.

We write $q_1 \equiv q_2$ if the states q_1 and q_2 are equivalent. Our last characterization for the language of a monoidal finite-state automaton uses the following notions.

Definition 2.1.19 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ be as above. For $p \in Q$ and $w \in M$, the set of *w-successors of p* is

$$Suc_{\mathcal{A}}(p, w) := \{q \in Q \mid \langle p, w, q \rangle \in \Delta^*\}.$$

For $P \subseteq Q$ the set of *w-successors of P* is

$$Suc_{\mathcal{A}}(P, w) := \bigcup_{p \in P} Suc_{\mathcal{A}}(p, w).$$

The set of *active states* for input $w \in M$ is $Act_{\mathcal{A}}(w) = Suc_{\mathcal{A}}(I, w)$.

Clearly an element $w \in M$ belongs to $L(\mathcal{A})$ iff the set of active states for input w contains a final state and $L(\mathcal{A}) = \{w \in M \mid Act_{\mathcal{A}}(w) \cap F \neq \emptyset\}$.

Definition 2.1.20 Let $\mathcal{A}' = \langle \mathcal{M}, Q', I', F', \Delta' \rangle$ and $\mathcal{A}'' = \langle \mathcal{M}, Q'', q_0'', F'', \Delta'' \rangle$ be two monoidal finite-state automata. $\mathcal{A}'$ is *isomorphic* to $\mathcal{A}''$ by the *state renaming function* $f : Q' \to Q''$ if f is a bijection such that

- $f(I') = I''$,
- $f(F') = F''$, and
- $\Delta'' = \{\langle f(p), m, f(q)\rangle \mid \langle p, m, q\rangle \in \Delta'\}$.

Intuitively, two automata are isomorphic iff they are identical up to a renaming of states. Clearly isomorphic automata are always equivalent.

Connections between classical finite-state automata and monoidal finite-state automata. Though the concept of a classical finite-state automaton is much more restricted than the general notion of a monoidal finite-state automaton, both can be related using the concept of a homomorphism.

Definition 2.1.21 Let $\mathcal{M}_1 = \langle M_1, \circ, e_1\rangle$ and $\mathcal{M}_2 = \langle M_2, \bullet, e_2\rangle$ be monoids, let $h : M_1 \to M_2$ be a monoid homomorphism. If $\mathcal{A}_1 = \langle \mathcal{M}_1, Q, I, F, \Delta_1\rangle$ is a monoidal finite-state automaton over $\mathcal{M}_1$, the automaton

$$\mathcal{A}_2 := \langle \mathcal{M}_2, Q, I, F, \{\langle p, h(a), q\rangle \mid \langle p, a, q\rangle \in \Delta_1\}\rangle$$

is called the *homomorphic image of* $\mathcal{A}_1$ *under* h.

It is trivial to see that in the above situation we have $L(\mathcal{A}_2) = h(L(\mathcal{A}_1))$.

Theorem 2.1.22 *Any monoidal finite-state automaton is the homomorphic image of a classical finite-state automaton. Any monoidal automaton language can be obtained as a homomorphic image of a classical automaton language.*

Proof. Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ be a monoidal finite-state automaton over $\mathcal{M} = \langle M, \circ, e\rangle$. Let Σ be the finite alphabet

$$\Sigma := \{a_m \mid \exists\, \langle q_1, m, q_2\rangle \in \Delta\}.$$

Consider the function $h_0 : \Sigma \to M$ defined as $h_0(a_m) := m$. Following Proposition 1.5.14 there exists a unique extension of h_0 to a homorphism $h : \Sigma^* \to M$. Let $\mathcal{A}' = \langle \Sigma, Q, I, F, \Delta'\rangle$ be the classical finite-state automaton where

$$\Delta' = \{\langle q_1, a_m, q_2\rangle \mid \langle q_1, m, q_2\rangle \in \Delta\}.$$

Clearly, $\mathcal{A}$ is a homomorphic image of $\mathcal{A}'$, $L(\mathcal{A}')$ is a classical automaton language and we have $h(L(\mathcal{A}')) = L(\mathcal{A})$. □

Definition 2.1.23 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ be a monoidal finite-state automaton. The classical finite-state automaton $\mathcal{A}' = \langle \Sigma, Q, I, F, \Delta'\rangle$ defined in the above proof is called the *free companion* of $\mathcal{A}$.

2.2 Closure properties of monoidal finite-state automata

Many interesting monoidal languages are obtained from combining simpler languages in a systematic way. This explains the interest in methods for combining monoidal finite-state automata. We show that the class of monoidal automaton languages is closed under four operations for monoidal languages mentioned in Section 1.5.

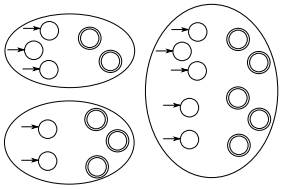

Figure 2.5: Union of two monoidal automata. The two input automata are shown on the left side, the result ist shown on the right side. We assume that the sets of states of the two input automata are disjoint.

Proposition 2.2.1 *The class of monoidal automaton languages is closed under monoid homomorphisms. The class of monoidal automaton languages over a given monoid $\mathcal{M}$ is closed under the regular operations union, monoidal product, and monoidal Kleene-Star.*

Proof. We first consider closure under monoid homomorphisms. Let $\mathcal{A}_1 = \langle \mathcal{M}, Q_1, I_1, F_1, \Delta_1 \rangle$ be a monoidal finite-state automaton, let $h : \mathcal{M} \to \mathcal{M}'$ be a monoid homomorphism. Define $\mathcal{A}'$ as the homomorphic image of $\mathcal{A}_1$ under h. Then we have $L(\mathcal{A}') = h(L(\mathcal{A}_1))$ (s.a.). To prove the second part, let

$$\begin{aligned} \mathcal{A}_1 &= \langle \mathcal{M}, Q_1, I_1, F_1, \Delta_1 \rangle \\ \mathcal{A}_2 &= \langle \mathcal{M}, Q_2, I_2, F_2, \Delta_2 \rangle \end{aligned}$$

be two monoidal finite-state automata over the same monoid $\mathcal{M}$. We may assume that $Q_1 \cap Q_2 = \emptyset$. It is simple to see that the following properties hold:

1. (Union) For the monoidal finite-state automaton

   $$\mathcal{A} = \langle \mathcal{M}, Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \rangle$$

   we have $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. The union is illustrated in Figure 2.5.

2. (Monoidal product) For the monoidal finite-state automaton

   $$\mathcal{A} = \langle \mathcal{M}, Q_1 \cup Q_2, I_1, F_2, \Delta_1 \cup \Delta_2 \cup \{\langle q_1, e, q_2 \rangle \mid q_1 \in F_1, q_2 \in I_2\} \rangle$$

   we have $L(\mathcal{A}) = L(\mathcal{A}_1) \circ L(\mathcal{A}_2)$. The monoidal product is illustrated in Figure 2.6.

3. (Monoidal Kleene-Star) Let q_0 be a new state, let

   $$\Delta := \Delta_1 \cup \{\langle q_0, e, q_1 \rangle \mid q_1 \in I_1\} \cup \{\langle q_2, e, q_0 \rangle \mid q_2 \in F_1\}.$$

   For the monoidal finite-state automaton

   $$\mathcal{A} = \langle \mathcal{M}, Q_1 \cup \{q_0\}, \{q_0\}, F_1 \cup \{q_0\}, \Delta \rangle$$

   we have $L(\mathcal{A}) = L(\mathcal{A}_1)^*$. The monoidal Kleene-star is illustrated in Figure 2.7.

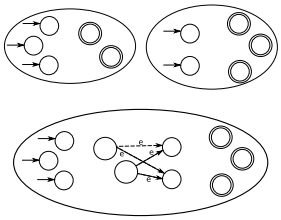

Figure 2.6: Monoidal product of two monoidal automata. The two input automata are shown above the result. Since the sets of states of the two input automata are disjoint the new transitions with label e (dashed arrows) represent the only connection.

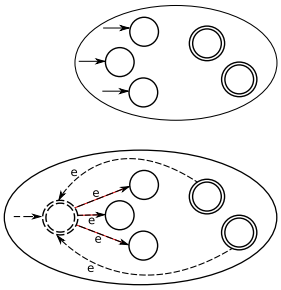

Figure 2.7: Kleene-star of a monoidal automaton. The new start state and the new transitions (label e) are marked using dashed lines.

□

For classical automata the above constructions can be generalized in the sense that the input automata may come with distinct input alphabets. For the output automaton we use the union of the two input alphabets. Classical finite-state automata have additional closure properties, see Section 3.2.

Remark 2.2.2 An alternative way to prove closure of monoidal automaton languages under the regular operations union product/concatenation and Kleene-Star is to show this property first for the case of free monoids, and then to use the first part, Proposition 1.5.12 and the fact that monoidal finite-state automata can be represented as homomorphic images of classical finite-state automata (Theorem 2.1.22). If two monoidal automata $\mathcal{A}_1$ and $\mathcal{A}_2$ as above are given, we represent $\mathcal{A}_i$ as the homomorphic image of a classical finite-state automaton $\mathcal{A}'_i$ under a homomorphism h_i $(i = 1, 2)$. We may assume that the alphabets Σ_1 and Σ_2 of $\mathcal{A}'_1$ and $\mathcal{A}'_2$ are disjoint. Then h_1 and h_2 can be extended to a unique homomomorphism $h : (\Sigma_1 \cup \Sigma_2)^* \to M$. Given a classical finite-state automaton for the union (concatenation) of the languages of $\mathcal{A}'_1$ and $\mathcal{A}'_2$, the homomorphic image under h represents a monoidal finite-state automaton recognizing the union (product) of $L(\mathcal{A}_1)$ and $L(\mathcal{A}_2)$.

2.3 Monoidal regular languages and monoidal regular expressions

Monoidal regular languages are defined using a simple induction. The empty language and the "singleton languages" containing exactly one monoid element are defined to be regular. Inductive rules then close the set of monoidal regular languages under union, monoidal product, and Kleene star. We shall see that the monoidal regular languages are exactly the monoidal automaton languages. This yields another, less procedural characterization of monoidal automaton languages.

Definition 2.3.1 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. We define the class of *monoidal regular languages* over $\mathcal{M}$ by induction:

1. $\emptyset$ is a monoidal regular language over $\mathcal{M}$;
2. if $m \in M$, then $\{m\}$ is a monoidal regular language over $\mathcal{M}$;
3. if $L_1, L_2 \subseteq M$ are monoidal regular languages over $\mathcal{M}$, then
   - $L_1 \cup L_2$ is a monoidal regular language over $\mathcal{M}$ (union),
   - $L_1 \circ L_2$ is a monoidal language over $\mathcal{M}$ (monoidal product, cf. Def. 1.5.5),
   - L_1^* is a monoidal regular language over $\mathcal{M}$ (monoidal Kleene star, cf. Def. 1.5.6).
4. There are no other monoidal regular languages over $\mathcal{M}$.

Definition 2.3.2 Let L be a monoidal regular language over the monoid $\mathcal{M}$. If $\mathcal{M}$ is the free monoid over a finite alphabet Σ, then L is called a *classical regular language*.

Remark 2.3.3 Since each monoid element m can be considered as a letter a_m of an alphabet, an induction using Proposition 1.5.12 shows that each monoidal regular language is the homomorphic image of a classical regular language. At the induction steps for union and product we may assume that L_1 and L_2 are homomorphic images $h_1(L_1')$ and $h_2(L_2')$ of classical regular languages L_1' and L_2' over free monoids *with disjoint alphabets*. Then there exists a unique homomorphism h_{12} for the free monoid over the joined alphabet that extends both h_1 and h_2. We have $h_{12}(L_1' \cup L_2') = L_1 \cup L_2$ and $h_{12}(L_1' \circ L_2') = L_1 \circ L_2$.

Definition 2.3.4 A *monoidal regular n-relation* is a monoidal regular language over a monoid $\mathcal{M}$ which is a Cartesian product of n monoids.

As in the classical case, monoidal regular languages can be represented by means of special expressions.

Definition 2.3.5 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. A *monoidal regular expression* over $\mathcal{M}$ for $M \cap \{(,), *, +, \cdot, \emptyset\} = \emptyset$ is a word over $M \cup \{(,), *, +, \cdot, \emptyset\}$.The set of monoidal regular expressions over $\mathcal{M}$ is defined by induction:

1. $\emptyset$ is a monoidal regular expression over $\mathcal{M}$;
2. if $m \in M$, then m is a monoidal regular expression over $\mathcal{M}$;
3. if E_1 and E_2 are monoidal regular expressions over $\mathcal{M}$, then
   - $(E_1 + E_2)$ is a monoidal regular expression over $\mathcal{M}$,
   - $(E_1 \cdot E_2)$ is a monoidal regular expression over $\mathcal{M}$,
   - (E_1^*) is a monoidal regular expression over $\mathcal{M}$.
4. There are no other monoidal regular expressions over $\mathcal{M}$.

As usual, for each monoidal regular expression E over $\mathcal{M}$ we inductively define a monoidal language $L(E)$, using the clauses

$$\begin{aligned} L(\emptyset) &:= \emptyset, \\ L(m) &:= \{m\} \quad (m \in M), \\ L(E_1 + E_2) &:= L(E_1) \cup L(E_2), \\ L(E_1 \cdot E_2) &:= L(E_1) \circ L(E_2), \\ L(E^*) &:= L(E)^*. \end{aligned}$$

This defines a natural correspondence between monoidal regular expressions and monoidal regular languages: obviously, the language of each monoidal regular expression is a monoidal regular language, and conversely each monoidal regular language can be represented by a monoidal regular expression.

If $\mathcal{M}$ is the free monoid over a finite alphabet Σ, then monoidal regular expressions are called *classical regular expressions*. In this case the second clauses in Definitions 2.3.1 and 2.3.5 can be refined, replacing arbitrary monoid elements m by letters $\sigma \in \Sigma$.

2.4 Equivalence between monoidal regular languages and monoidal automaton languages

In this section we show that monoidal regular expressions and monoidal finite-state automata yield two descriptions of the same class of languages. Our first proposition shows that the simplest monoidal regular languages can be represented by means of monoidal finite-state automata.

Proposition 2.4.1

1. *(Empty language) For* $\mathcal{A}_\emptyset = \langle \mathcal{M}, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ *we have* $L(\mathcal{A}_\emptyset) = \emptyset$.
2. *(Single element languages) Let* $m \in M$. *For the monoidal finite-state automaton* $\mathcal{A}_m = \langle \mathcal{M}, \{q_0, q_1\}, \{q_0\}, \{q_1\}, \{\langle q_0, m, q_1 \rangle\} \rangle$ *we have* $L(\mathcal{A}_m) = \{m\}$.

The following theorem presents a deeper result for the correspondence between monoidal automaton languages and regular languages.

Theorem 2.4.2 (Kleene) *A monoidal language is regular if and only if it is a monoidal automaton language.*

Proof. ("⇒") This direction follows directly from Propositions 2.4.1 and the closure properties given in Proposition 2.2.1.

("⇐") Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal finite-state automaton. Let $n := |Q|$, let $\langle q_1, \ldots, q_n \rangle$ be a fixed enumeration of the states in Q. For $0 \leq k \leq n$ let $Q_k := \{q_1, \ldots, q_k\}$. For $i, j, k \in \{1, \ldots, n\}$ let R_{ij}^k denote the set of all labels of paths π starting in q_i and ending q_j for which all intermediate states (beside the beginning and the ending states) are in Q_k. For $i = j$ the set R_{ij}^k includes the neutral element e, which is the label of the null path of q_i. Each set R_{ij}^k is a monoidal language. Note that there are no intermediate states for $k = 0$ since $Q_0 = \emptyset$. Thus

$$R_{ij}^0 = \begin{cases} \{m \in M \mid \langle q_i, m, q_j \rangle \in \Delta\} & \text{if } i \neq j \\ \{e\} \cup \{m \in M \mid \langle q_i, m, q_j \rangle \in \Delta\} & \text{if } i = j. \end{cases}$$

Each set R_{ij}^0 - as a finite collection of monoid elements - is a regular monoidal language. For $1 \leq i \leq n$ we have

$$R_{ij}^k = R_{ij}^{k-1} \ \cup \ (R_{ik}^{k-1} \circ (R_{kk}^{k-1})^* \circ R_{kj}^{k-1})$$

The expression takes into account paths from q_i to q_j with intermediate states in Q_{k-1} and in addition paths with some visits of the intermediate state q_k. See Figure 2.8 for an illustration. From the above presentation - which uses the monoidal operations union, product, and Kleene star - it follows by induction over k that for any $i, j, k \in \{0, \ldots, n\}$ the monoidal languages R_{ij}^k are regular. Since

$$L(\mathcal{A}) = \bigcup_{q_i \in I, q_j \in F} R_{ij}^n$$

it follows that $L(\mathcal{A})$ is a monoidal regular language. □

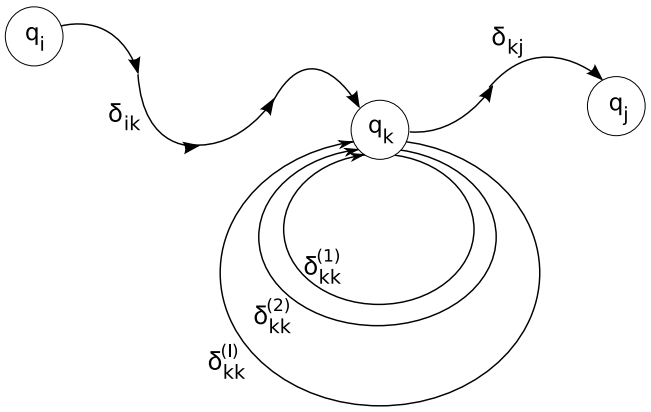

Figure 2.8: Illustration for the above proof of Theorem 2.4.2. The expression $R_{ik}^{k-1} \circ (R_{kk}^{k-1})^* \circ R_{kj}^{k-1}$ describes paths that first lead from q_i to q_k, only visiting states in $\{q_1, \ldots, q_{k-1}\}$, with label $\delta_{i,k}$ in R_{ik}^{k-1}, then a sequence of similar $l \geq 0$ subpaths from q_k to q_k with labels $\delta_{k,k}^{(1)}, \ldots \delta_{k,k}^{(l)}$ in R_{kk}^{k-1}, and finally a similar path from q_k to q_j with label $\delta_{k,j}$ in R_{kj}^{k-1}. In the figure, states on the subpaths that are not explicitly shown are always in $\{q_1, \ldots, q_{k-1}\}$.

Remark 2.4.3 The above general form of the Kleene Theorem can be considered as a direct consequence of the fact that the theorem holds for the special case of free monoids. In fact, an alternative way to prove Theorem 2.4.2 would be to first prove the theorem for the special case of free monoids, and then to use Remark 2.3.3: given a monoidal automaton $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ we may represent it as the homomorphic image of a classical automaton $\mathcal{A}'$. Once we have constructed a classical regular expression α' such that $L(\alpha') = L(\mathcal{A}')$ we may translate α' into a monoidal regular expression α such that $L(\alpha) = L(\mathcal{A})$. Details are left to the reader.

2.5 Simplifying the structure of monoidal finite-state automata

In contrast to the operations considered in Section 2.2 the operations considered here do not modify the monoidal language of the given automaton. The goal is rather to simplify automata from a structural point of view, thus facilitating the recognition process.

Trimming. Monoidal finite-state automata sometimes contain states that are "useless" in the sense that deleting those states with their transitions will not change the language. A state is useless in this sense whenever it does not belong to a successful path.

Definition 2.5.1 A monoidal finite-state automaton $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ is *trimmed* iff each state $q \in Q$ is on a successful path of $\mathcal{A}$.

It is straightforward to check for a given state $q \in Q$ of a given monoidal

finite-state automaton $\mathcal{A}$ if it is on a successful path. In the negative case we may delete q and all transitions leading to or departing from q. In this way, a trimmed monoidal finite-state automaton $\mathcal{A}'$ equivalent to $\mathcal{A}$ is obtained. For details we refer to Algorithm 8.1.6.

Removal of e-transitions. In general, complex procedures are needed to see if a word or monoid element belongs to the language of a finite-state automaton. The following definition captures a structural restriction that leads to a simpler acceptance procedure.

Definition 2.5.2 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal finite-state automaton where $\mathcal{M} = \langle M, \circ, e \rangle$. $\mathcal{A}$ is called *e-free* iff $\Delta \subseteq Q \times (M \setminus \{e\}) \times Q$.

We next show how to convert a given monoidal finite-state automaton to an equivalent e-free monoidal finite-state automaton. There are several ways to proceed. The following notions are central.

Definition 2.5.3 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal finite-state automaton. The *forward e-closure* $C_e^f : Q \to 2^Q$ is defined as

$$C_e^f(q) = \{q' \in Q \mid \langle q, e, q' \rangle \in \Delta^*\}.$$

The *backward e-closure* $C_e^b : Q \to 2^Q$ is defined as

$$C_e^b(q) = \{q' \in Q \mid \langle q', e, q \rangle \in \Delta^*\}.$$

We now present two constructions for removal of e-transitions. In both cases we do not modify the set of states. The first construction is based on the forward e-closure.

Proposition 2.5.4 *For any monoidal finite-state automaton $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ there exists an equivalent e-free monoidal finite-state automaton $\mathcal{A}$' with the same set of states.*

Proof. The monoidal finite-state automaton

$$\mathcal{A}' = \left\langle \mathcal{M}, Q, C_e^f(I), F, \Delta' \right\rangle,$$

where

$$\Delta' = \{\langle q_1, a, q_2 \rangle \mid \exists \langle q_1, a, q' \rangle \in \Delta : q_2 \in C_e^f(q') \ \&\ a \neq e\}.$$

is e-free. If $w \in M$ is in $L(\mathcal{A})$, then there exists a path π from an initial state i to a final state with label w. If this path starts with some e-transitions, all states on this initial part are initial states of $\mathcal{A}'$. The remaining transitions in π (if any) can be split into subsequences where a transition of the form $\langle q_1, a, q' \rangle \in \Delta$ with $a \neq e$ is followed by a (possibly empty) sequence of e-transitions. Each subsequence corresponds to a single transition in $\mathcal{A}'$. Thus we have a path π' from an initial state i' to a final state with label w in $\mathcal{A}'$ and $w \in L(\mathcal{A}')$. Conversely, from each successful path π' in $\mathcal{A}'$ from $i' \in C_e^f(I)$ to a final state f we obtain a parallel path π in $\mathcal{A}'$ from an initial state $i \in I$ to f. It follows that $\mathcal{A}'$ is equivalent to $\mathcal{A}$. □

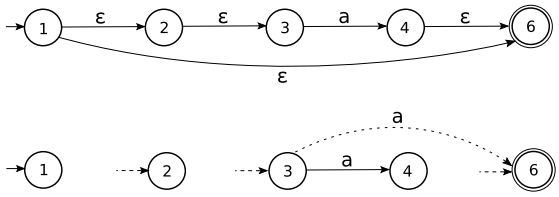

Figure 2.9: Construction of an ε-free automaton - removal of ε-transitions using forward ε-closures. Though the language of the automaton is not affected, the language of several states is changed.

Example 2.5.5 The construction based on forward e-closure is illustrated in Figure 2.9. A classical input automaton $\mathcal{A}$ is shown in the upper part. The lower part shows the resulting automaton $\mathcal{A}'$. All ε-transitions are removed. New transitions added are shown using dashed lines. In order to preserve the language of the automaton, new initial states need to be added.

A symmetric construction based on backward e-closure where the set of final states is extended is omitted. Though the above construction yields an automaton accepting the same language, it has the disadvantage that the language of a given state can be changed. For example, in Figure 2.9 the language of states 1, 2, and 4 becomes empty after the transformation. We now present an alternative construction that preserves the language of each state, which is useful for later constructions. The price is that we have to add more transitions. Here we use both forward and backward e-closures.

Proposition 2.5.6 *For any monoidal finite-state automaton $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ there exists an equivalent e-free monoidal finite-state automaton $\mathcal{A}$' with the same set of states such that for each $q \in Q$ we have $L_{\mathcal{A}}(q) = L_{\mathcal{A}'}(q)$.*

Proof. Consider the monoidal finite-state automaton

$$\mathcal{A}' = \left\langle \mathcal{M}, Q, I, C_e^b(F), \Delta' \right\rangle,$$

where

$$\Delta' = \{\langle q_1', m, q_2' \rangle \mid \exists \langle q_1, m, q_2 \rangle \in \Delta : q_1' \in C_e^b(q_1) \;\&\; q_2' \in C_e^f(q_2) \;\&\; m \neq e\}.$$

Let $q \in Q$. First, let $m \in L_{\mathcal{A}}(q)$. Then there exists a path π of $\mathcal{A}$ leading from q to a final state f with label m. Clearly, if π contains at least one transition with a label $m' \neq e$, then we can build in $\mathcal{A}'$ a path π' with label w from q to f using the transitions in Δ'. If $m = e$ and all transitions in π have label e, then $q \in C_b(f)$, $q \in F' := C_e^b(F)$ and again $e \in L_{\mathcal{A}'}(q)$. This shows that $L_{\mathcal{A}}(q) \subseteq L_{\mathcal{A}'}(q)$. Conversely, from each successful path π' in $\mathcal{A}'$ from p to a final state f we obtain a parallel path π in $\mathcal{A}'$ from p to f. From f, using a sequence of e-transitions we reach a final state f' in F. □

As an illustration consider Figure 2.10. A discussion of distinct constructions for removal of e-transitions and their computational properties can be found in [van Noord, 2000].

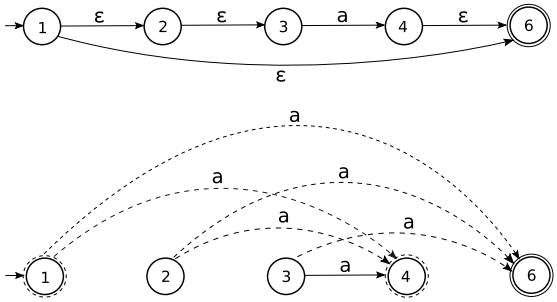

Figure 2.10: Construction of an ε-free automaton - removal of ε-transitions using forward and backward ε-closures. New transitions added are shown using dashed lines. The language of each state is preserved. To this end, states 1 and 4 become final.

Remark 2.5.7 The e-removal only makes sense in situations where the underlying monoid does not contain zero divisors, i.e., for monoids where for all $m_1, m_2 \in M$ we have $m_1 \neq e \;\&\; m_2 \neq e \to m_1 \circ m_2 \neq e$. The most important examples are free monoids and Cartesian products of free monoids. If the underlying monoid does not contain zero divisors, then every proper path of an e-free automaton has a label different from e.

Removal of complex transition labels for automata over a free monoid. An important subcase are finite-state automata over the free monoid for an alphabet Σ. In this situation we often want to avoid complex transition labels, i.e., words of length ≥ 2. Note that we then obtain a classical finite-state automaton in the sense of Definition 2.1.3. The following construction adds new states, preserving the language of "old" states.

Proposition 2.5.8 *Let $\mathcal{A}$ be a monoidal finite-state automaton over a free monoid $\mathcal{M} = \Sigma^*$. Then $\mathcal{A}$ can be converted to a classical finite-state automaton $\mathcal{A}'$ with a set of states Q' such that $Q \subseteq Q'$ and for each $q \in Q$ we have $L_{\mathcal{A}}(q) = L_{\mathcal{A}'}(q)$.*

Proof. The automaton $\mathcal{A}'$ is obtained by introducing intermediate states for each label consisting of more than one symbol: we substitute each transition

$$t = \langle q', a_1 a_2 \dots a_n, q'' \rangle \in \Delta$$

where $n > 1$ with a sequence of transitions of the form

$$\left\langle q', a_1, q_1^t \right\rangle, \left\langle q_1^t, a_2, q_2^t \right\rangle, \dots, \left\langle q_{n-1}^t, a_n, q'' \right\rangle$$

where $q_1^t, q_2^t, \dots, q_{n-1}^t$ are new non-final states (the e_i representing the neutral elements of the monoids). Obviously the set of states Q is extended and the language of states $q \in Q$ is not modified. □

2.6 Summing up

Generalizing the concept of a classical finite-state automaton we introduced the notion of a monoidal finite-state automaton. We developed a rudimentary theory for monoidal finite-state automata and the associated languages, proving closure under the regular operations union, monoidal product, and monoidal Kleene-Star. After introducing monoidal regular languages, a generalized version of the classical Kleene Theorem on the equivalence between regular languages and finite-state automata for the monoidal case was given. Remarks 2.2.2, 2.3.3 and 2.4.3 show that all these results for the monoidal case can be seen as consequences of the fact that each monoid can be described as the homomorphic image of a free monoid.

2.7 Exercises for Chapter 2

Exercise 2.7.1 Show that for each monoidal finite-state automaton $\mathcal{A}$ there exists an equivalent monoidal finite-state automaton with exactly one initial state and exactly one final state.

Exercise 2.7.2 Let $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ be a classical finite-state automaton with a non-empty set of final states. Assume that there exists a number n such that $|L_{\mathcal{A}}(q)| = n$ for each $q \in Q$. Show that $|L_{\mathcal{A}}| \leq |I|$.

Exercise 2.7.3 In the proof of Proposition 2.2.1 we assumed that $Q_1 \cap Q_2 = \emptyset$. Would the following statements remain correct if $Q_1 \cap Q_2 \neq \emptyset$? If you think that a statements becomes wrong, give a (counter-)example.

Exercise 2.7.4 Consider the classical finite-state automaton $\mathcal{A}$ shown in Figure 2.2. Following the proof of Theorem 2.4.2 compute the sets $R^i_{j,l}$ for $0 \leq i \leq 3$ and $1 \leq j, l \leq 3$. Use this to compute $L(\mathcal{A})$.

Exercise 2.7.5 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal finite-state automaton. Let $C^f_e : Q \to 2^Q$ denote the forward e-closure. Let $f_1, f_2 \in F$ be two distinct final states such that $f_2 \in C^f_e(f_1)$. Show that $\mathcal{A}' := \langle \mathcal{M}, Q, I, F \setminus \{f_1\}, \Delta \rangle$ is equivalent to $\mathcal{A}$. Let $q \in Q \setminus F$ be a nonfinal state such that $f_2 \in C^f_e(q)$. Show that $\mathcal{A}'' := \langle \mathcal{M}, Q, I, F \cup \{q\}, \Delta \rangle$ is equivalent to $\mathcal{A}$. Using these observations, describe general operations for modifying the set of final states that do not affect the language accepted.

Exercise 2.7.6 As in [Lewis and Papadimitriou, 1998], *arithmetic progressions* are defined as sets of the form $\{p + qn \mid n \in \mathbb{N}\}$ for some $p, q \in \mathbb{N}$. We consider languages L over the single-letter alphabet $\{a\}$.

1. Show that if $\{n \in \mathbb{N} \mid a^n \in L\}$ is an arithmetic progression, then L is regular.
2. Show that if $\{n \in \mathbb{N} \mid a^n \in L\}$ is a union of finitely many arithmetic progressions, then L is regular.
3. (Hard) Show that if L is regular, then $\{n \in \mathbb{N} \mid a^n \in L\}$ is a union of finitely many arithmetic progressions.

Chapter 3

Classical finite-state automata and regular languages

Classical finite-state automata represent the most important class of monoidal finite-state automata. Since the underlying monoid is free, this class of automata has several interesting specific features. Many textbooks consider exclusively the classical case (e.g. [Hopcroft et al., 2006, Kozen, 1997, Lewis and Papadimitriou, 1998]). We see below that each classical finite-state automaton can be converted to an equivalent classical finite-state automaton where the transition relation is a function. This form of "deterministic" automata offers a very efficient recognition mechanism since each input word is consumed on at most one path. The fact that each classical finite-state automaton can be converted to a deterministic automaton can be used to show that the class of languages that can be recognized by a classical finite-state automaton is closed under intersections, complements, and set differences. The characterization of regular languages and deterministic finite-state automata in terms of the "Myhill-Nerode equivalence relation" to be introduced below offers an algebraic view on these notions and leads to the concept of minimal deterministic automata.

3.1 Deterministic finite-state automata

Deterministic finite-state devices are characterized by the property that with any input at most one state can be reached. Deterministic finite-state automata represent the simplest form.

Definition 3.1.1 Let $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ be a classical finite-state automaton. $\mathcal{A}$ is *deterministic* iff the following conditions hold:

- $\mathcal{A}$ has exactly one initial state q_0,
- all transition labels are letters in Σ and
- $\langle p, a, q \rangle \in \Delta$ and $\langle p, a, q' \rangle \in \Delta$ implies $q = q'$ for all triples in Δ.

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

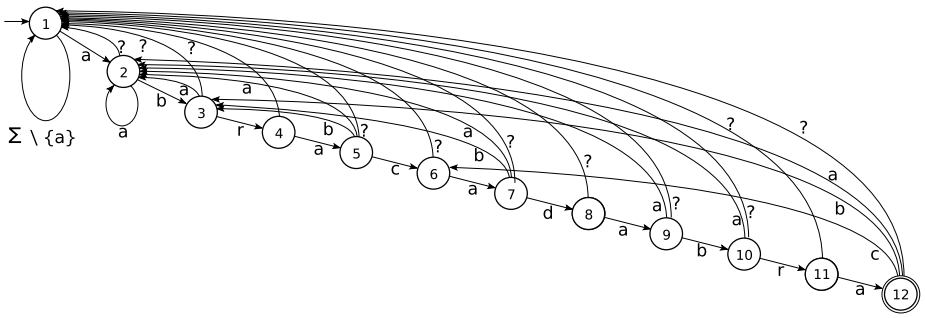

Figure 3.1: A deterministic classical finite-state automaton for searching occurrences of the pattern *abracadabra* in a text, cf. Examples 2.1.5 and 3.1.2.

In this situation the transition relation can be described as a (partial) function $\delta : Q \times \Sigma \to Q$.

Deterministic finite-state automata are often represented in the form

$$\langle \Sigma, Q, q_0, F, \delta \rangle$$

using the start state (instead of the singleton set) as third component and representing the transition relation as a partial function. The importance of the concept of determinism relies on the fact that the "recognition problem" for a classical automaton $\mathcal{A}$ - the problem to decide if a given input string w belongs to the language of $\mathcal{A}$ (s.b.) - can be solved very efficiently - in time $O(|w|)$ - if $\mathcal{A}$ is deterministic. We did not introduce this concept for monoidal finite-state automata since determinization of monoidal finite-state automata is not possible in general (see Section 3.7). There are however distinct options in some special cases, which will be introduced in Chapter 5.

Example 3.1.2 Figure 3.1 shows a deterministic variant of the classical finite-state automaton presented in Example 2.1.5, see Figure 2.3. Outgoing transitions from a state q with the dummy label "?" are used for all letters σ of the alphabet Σ where q does not have another outgoing transition with letter σ. Hence the automaton has a total transition function and the automaton can be used for finding occurrences of the pattern *abracadabra* in any text over the alphabet Σ. When reading a text $t \in \Sigma^*$, the automaton is in state 12 always after having read an occurrence of *abracadabra* in t.

Remark 3.1.3 If $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ is a deterministic classical finite-state automaton, then the generalized transition relation can be described as a (partial) function $\delta^* : Q \times \Sigma^* \to Q$. It is called the *generalized transition function*.

The trie for a finite set of words. We finish the section with an important special case of a classical deterministic finite-state automaton.

Definition 3.1.4 Let $D \subseteq \Sigma^*$ be a finite set of words over the alphabet Σ. The *trie* for D is the deterministic finite-state automaton

$$\mathcal{A}_D = \langle \Sigma, Pref(D), \varepsilon, D, \delta \rangle$$

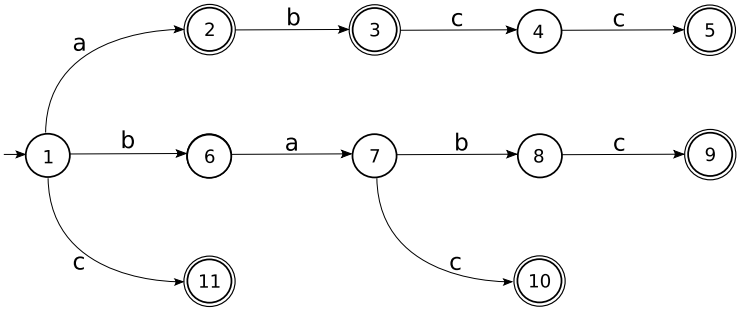

Figure 3.2: The trie for $\mathcal{D} = \{a, ab, abcc, babc, bac, c\}$.

where

$$\delta = \{\langle \alpha, \sigma, \alpha \cdot \sigma \rangle \mid \sigma \in \Sigma \ \& \ \alpha, \alpha \cdot \sigma \in \mathit{Pref}(D)\}.$$

The proof of the following proposition is obvious.

Proposition 3.1.5 *Let $D \subseteq \Sigma^*$ be a finite set of words. Then*

1. *$L(\mathcal{A}_D) = D$,*
2. *Each state q of $\mathcal{A}_D$ can be reached on exactly one path from the initial state ε. The label of this path is the string q.*

In Definition 3.1.4, using the prefixes of strings in D as names for the states of the trie is elegant from a mathematical point of view. However, in praxis often other state names (e.g., $1, \ldots, n$) are used. In this sense "the" trie for a finite set of strings is only fixed up to isomorphism, i.e., renaming of states. When using a version of the trie where there is no connection between state names and prefixes, the following notion is useful.

Definition 3.1.6 Let $\mathcal{A}_D = \langle \Sigma, Q, q_0, F, \delta \rangle$ represent the trie for the finite set $D \subseteq \Sigma^*$, let $q \in Q$. The label of the unique path π from q_0 to q is called the *path label* for q and denoted $\mathit{plab}(q)$. The length of π is called the *depth* of q and denoted $d(q)$.

Example 3.1.7 Figure 3.2 shows the trie for $\mathcal{D} = \{a, ab, abcc, babc, bac, c\}$. For State 4 we have $d(4) = 3$ and $\mathit{plab}(4) = abc$.

3.2 Determinization of classical finite-state automata

Our next aim is to show that each classical finite-state automaton can be effectively converted to an equivalent deterministic classical finite-state automaton. As a preliminary step, combining Propositions 2.5.6 and 2.5.8 we obtain

Proposition 3.2.1 *Every monoidal finite-state automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ over the free monoid Σ^* can be converted to an equivalent classical finite-state*

automaton $\mathcal{A}'$ without ε-transitions with a set of states Q' such that $Q \subseteq Q'$ and for each $q \in Q$ we have $L_{\mathcal{A}}(q) = L_{\mathcal{A}'}(q)$.

The proof of the following theorem provides an effective procedure for determinization of classical finite-state automata.

Theorem 3.2.2 (Determinization of classical finite-state automata) *Every classical finite-state automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ can be converted to an equivalent deterministic finite-state automaton $\mathcal{A}_D$ with total transition function.*

Proof. Using Proposition 3.2.1 we may assume without loss of generality that $\mathcal{A}$ does not have ε-transitions and no transition labels with more than one symbol, which means that $\Delta \subseteq Q \times \Sigma \times Q$. Consider the automaton

$$\mathcal{A}_D := \left\langle \Sigma, 2^Q, I, F_D, \delta \right\rangle$$

where $F_D := \{S \subseteq Q \mid S \cap F \neq \emptyset\}$ and

$$\delta(S, a) := \{q \in Q \mid \exists q_1 \in S : \langle q_1, a, q \rangle \in \Delta\}$$

for $S \subseteq Q, a \in \Sigma$. Clearly δ is a total function $2^Q \times \Sigma \to 2^Q$ and $\mathcal{A}_D$ has exactly one initial state I. This shows that $\mathcal{A}_D$ is a deterministic finite-state automaton with total transition function δ. We claim that for all $w \in \Sigma^*$ and all $p \in Q$ the following properties are equivalent:

1. $\exists i \in I : (i, w, p) \in \Delta^*$,
2. $p \in \delta^*(I, w)$.

For the proof we use induction on $n := |w|$. For $n = 0$ we have $w = \varepsilon$. Since $\mathcal{A}$ does not have ε-transitions, $\exists i \in I : (i, \varepsilon, p) \in \Delta^*$ iff $p \in I = \delta^*(I, \varepsilon)$. For the induction step, let $w = w'\sigma$ denote a word of length $n + 1$ where $\sigma \in \Sigma$. Assume that $\exists i \in I : (i, w'\sigma, p) \in \Delta^*$. The properties of Δ show that there exists $p' \in Q$ such that $(i, w', p') \in \Delta^*$ and $(p', \sigma, p) \in \Delta$. The induction hypothesis shows that $p' \in \delta^*(I, w') =: S$. It follows from the definition of δ that $p \in \delta^*(I, w)$. For the converse direction assume that $p \in \delta^*(I, w'\sigma)$. The definition of δ implies that $S := \delta^*(I, w')$ contains a state $p' \in Q$ such that $(p', \sigma, p) \in \Delta$. The induction hypothesis shows $\exists i \in I : (i, w', p) \in \Delta^*$, which implies that $\exists i \in I : (i, w, p) \in \Delta^*$. Hence the claim is proved.

Using the claim and the definition of F_D we have for any $w \in \Sigma^*$:

$$\begin{aligned} w \in L(\mathcal{A}) &\leftrightarrow \exists i \in I, f \in F : (i, w, f) \in \Delta^* \\ &\leftrightarrow \exists f \in F : f \in \delta^*(I, w) \\ &\leftrightarrow \delta^*(I, w) \in F_D \\ &\leftrightarrow w \in L(\mathcal{A}_D) \end{aligned}$$

Hence $\mathcal{A}$ and $\mathcal{A}_D$ are equivalent. □

Remark 3.2.3 It should be noted that the simple determinization procedure obtained from this proof can be considerably refined. The basic idea is to use

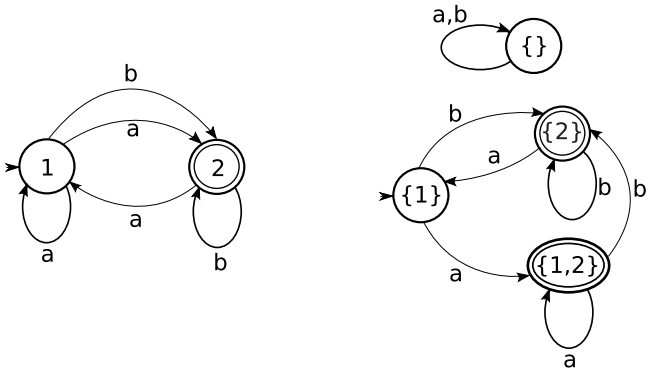

Figure 3.3: Determinization of a classical finite-state automaton (left-hand side) using the procedure described in the proof of Theorem 3.2.2. The resulting deterministic finite-state automaton (right-hand side) has one superfluous state that cannot be reached from the start state. When using the improved construction described in Remark 3.2.3 this state is not produced.

the collection of all sets of active states (cf. Definition 2.1.19) for arbitrary input of the given non-deterministic classical finite-state automaton as the new set of states of the deterministic finite-state automaton. In this case, the states of the deterministic automaton are exactly the subsets S of Q with the following property (†): there exists a string $w \in \Sigma^*$ such that S is the set of states that can be reached from a start state $i \in I$ on a path with label w.

Remark 3.2.4 In practice we may omit the explicit construction of the empty set, all subsets that are not reachable, and all transitions to and from those subsets, which often leads to a much smaller deterministic automaton with partial transition function. There are also determinization procedures that directly include removal of ϵ-transitions.

Example 3.2.5 The determinization of a classical finite-state automaton using the procedure described in the proof of Theorem 3.2.2 is illustrated in Figure 3.3.

3.3 Additional closure properties for classical finite-state automata

In the previous chapter we have seen that the class of languages accepted by monoidal finite-state automata is closed under homomorphic images, union, monoidal product, and monoidal Kleene-Star. Theorem 3.2.2 can be used to prove that the class of languages accepted by classical finite-state has additional closure properties. We start with a series of automaton constructions.

Proposition 3.3.1 (Complementing deterministic FSA) *Let*

$$\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$$

be a deterministic finite-state automaton where δ is a total function. Let

$$\mathcal{A}' = \langle \Sigma, Q, q_0, Q \setminus F, \delta \rangle .$$

Then $L(\mathcal{A}') = \Sigma^* \setminus L(\mathcal{A})$.

Proof. Using induction on $|w|$ we show that for each word $w \in \Sigma^*$ there exists a unique path in $\mathcal{A}$ (and thus in $\mathcal{A}'$) starting at q_0 with label w. For $w = \varepsilon$ the null path for q_0 is the unique path starting at q_0 with label w. For the induction step let $w = w'\sigma$ be a word of length $n + 1 > 0$ where $\sigma \in \Sigma$. By induction hypothesis there exists a unique path π' from q_0 to some state $q' \in Q$ with label w'. Since δ is a function and $\delta(q', \sigma)$ is defined, adding to π' the transition $q' \to^{\sigma} \delta(q', \sigma)$ we obtain a path π from q_0 with label $w'\sigma$. Uniqueness of π' implies uniqueness of π. The definition of the set of final states for $\mathcal{A}'$ shows that a path π starting at q_0 is successful in $\mathcal{A}$ iff it is not successful in $\mathcal{A}'$. Hence any word $w \in \Sigma^*$ is accepted in $\mathcal{A}$ iff it is not accepted in $\mathcal{A}'$. □

The next proposition introduces constructions for intersection, difference, and reversal of automaton languages.

Proposition 3.3.2 *Let $\mathcal{A}_1 = \langle \Sigma, Q_1, I_1, F_1, \Delta_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, I_2, F_2, \Delta_2 \rangle$ be two classical ε-free finite-state automata. Then the following holds:*

1. *(Intersection for ε-free classical finite-state automata) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are ε-free finite-state automata, then for the finite-state automaton*

   $$\mathcal{A} := \langle \Sigma, Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \Delta' \rangle$$

   where $\Delta' := \{\langle \langle q_1, q_2 \rangle, a, \langle r_1, r_2 \rangle \rangle \mid \langle q_1, a, r_1 \rangle \in \Delta_1 \ \& \ \langle q_2, a, r_2 \rangle \in \Delta_2\}$ we have $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

2. *(Difference for deterministic classical finite-state automata) If $\mathcal{A}_2$ is a deterministic classical finite-state automaton and the transition function of $\mathcal{A}_2$ is total, then for the finite-state automaton*

   $$\mathcal{A} := \langle \Sigma, Q_1 \times Q_2, I_1 \times I_2, F_1 \times (Q_2 \setminus F_2), \Delta' \rangle$$

   where $\Delta' := \{\langle \langle q_1, q_2 \rangle, a, \langle r_1, r_2 \rangle \rangle \mid \langle q_1, a, r_1 \rangle \in \Delta_1 \ \& \ \langle q_2, a, r_2 \rangle \in \Delta_2\}$ we have $L(\mathcal{A}) = L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$.

Proof.

1. Let $w = \sigma_1 \cdots \sigma_n \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ where $\sigma_i \in \Sigma$ $(1 \leq i \leq n)$. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are ε-free there exists successful paths

   $$\begin{aligned} \pi_1 : \quad & i_1 \to^{\sigma_1} q_{1,1} \ldots \to^{\sigma_n} q_{1,n} \\ \pi_2 : \quad & i_2 \to^{\sigma_1} q_{2,1} \ldots \to^{\sigma_n} q_{2,n} \end{aligned}$$

   in $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Note that $\langle i_1, i_2 \rangle \in I_1 \times I_2$ and $\langle q_{1,n}, q_{2,n} \rangle \in F_1 \times F_2$. Thus

   $$\pi : \quad \langle i_1, i_2 \rangle \to^{\sigma_1} \langle q_{1,1}, q_{2,1} \rangle \ldots \to^{\sigma_n} \langle q_{1,n}, q_{2,n} \rangle$$

   is a successful path with label w in $\mathcal{A}$. Conversely, if $w = \sigma_1 \cdots \sigma_n \in \mathcal{A}$, then there exists a successful path π in $\mathcal{A}$ with label w, which must have the above

form and can be decomposed into successful paths π_1 and π_2 of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively.
2. Let $w \in L(\mathcal{A})$. Then there exists a successful path π of $\mathcal{A}$ with label w. Using the first (second) projection of the states we obtain a successful path π_1 with label w of $\mathcal{A}_1$ and a parallel path π_2 of $\mathcal{A}_2$ leading from the start to a non-final state. Since $\mathcal{A}_2$ is deterministic it follows that $w \in L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$. Conversely, if $w \in L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$ we have a successful path with label w in $\mathcal{A}_1$. Since the transition function of $\mathcal{A}_2$ is total we have a parallel path from the start to a non-final state in $\mathcal{A}_2$. The paths can be combined to a successful path with label w in $L(\mathcal{A})$ as in the previous point. □

Remark 3.3.3 The construction used in Point 1 and 2 of the above proposition is called *Cartesian product of automata*. It can be used for an alternative construction of the union of two automaton languages and will be used in various other constructions in the next chapters.

Proposition 3.3.4 *Let $\mathcal{A} := \langle \Sigma^*, Q, F, I, \Delta \rangle$ be a monoidal finite-state automaton over the free monoid Σ^*. Then for the monoidal finite-state automaton*

$$\mathcal{A}' := \langle \Sigma^*, Q, F, I, \Delta' \rangle$$

where $\Delta' = \{\langle q_2, \rho(a), q_1 \rangle \mid \langle q_1, a, q_2 \rangle \in \Delta\}$ we have $L(\mathcal{A}') = \rho(L(\mathcal{A}))$.

Proof. Note that initial (final) states of $\mathcal{A}$ represent final (initial) states of $\mathcal{A}'$ and vice versa. Since labels in $\mathcal{A}$ and in $\mathcal{A}'$ are in Σ^*, when reading a successful path in $\mathcal{A}$ with label w in reverse order we obtain a successful path in $\mathcal{A}'$ for $\rho(w)$ and vice versa. □

From the above constructions, using Proposition 3.2.1 and Theorem 3.2.2 we immediately obtain the following result.

Corollary 3.3.5 *The class of languages accepted by classical finite-state automata is closed under complement, intersection, set difference, and reversal.*

3.4 Minimal deterministic finite-state automata and the Myhill-Nerode equivalence relation

Since the size of automata is often large, there is an obvious interest in finding equivalent small automata. In this part we describe a classical result on how to build for any classical automaton language L a deterministic finite-state automaton accepting L that is minimal with respect to the class of all deterministic finite-state automata recognizing L. Minimality only refers to deterministic automata, in general there can be non-deterministic automata for L with a smaller number of states.

In order to find a minimal deterministic automaton for a classical language it is important to take an algebraic perspective. In this section we describe deterministic finite-state automata recognizing a classical language L in terms of specific equivalence relations. Studying the spectrum of these equivalence relations will lead us to a minimal deterministic automaton for L. Let Σ be a finite alphabet.

Definition 3.4.1 An equivalence relation $R \subseteq \Sigma^* \times \Sigma^*$ is called *right invariant* if

$$\forall u, v \in \Sigma^* : u \; R \; v \rightarrow (\forall w \in \Sigma^* : u \cdot w \; R \; v \cdot w).$$

Definition 3.4.2 Let $L \subseteq \Sigma^*$ be a language, let $R \subseteq \Sigma^* \times \Sigma^*$ be an equivalence relation. L and R are called *compatible* if

$$\forall u, v \in \Sigma^* : (u \in L \; \& \; u \; R \; v) \rightarrow v \in L.$$

The condition means that each equivalence class $[v]_R$ is either a subset of L or we have $[v]_R \cap L = \emptyset$.

Example 3.4.3 Let $\Sigma := \{a, b\}$. Define two strings $u, v \in \Sigma^*$ to be equivalent (written $u \sim v$) if the numbers of occurrences of the letter a in u and v respectively are congruent modulo 3. Then $\sim$ is a right invariant equivalence relation since for each w the numbers of occurrences of the letter a in uw and vw respectively are again congruent mudulo 3. Let L be the set of all strings $u \in \Sigma^*$ where the number of occurrences of the letter a is divisible by 3. Then $\sim$ and L are compatible.

We now show that given a right invariant equivalence relation R with finite index that is compatible with a language L we may use R to define a classical finite-state automaton accepting the language L.

Proposition 3.4.4 *Let $R \subseteq \Sigma^* \times \Sigma^*$ be a right invariant equivalence relation such that the index of R is finite, let $L \subseteq \Sigma^*$ be a language over Σ compatible with R. Then for the deterministic classical finite-state automaton*

$$\mathcal{A}_{R,L} = \langle \Sigma, \{[s]_R \mid s \in \Sigma^*\}, [\varepsilon]_R, \{[s]_R \mid s \in L\}, \delta_R \rangle$$

with transition function $\delta_R = \{\langle [u]_R, a, [u \cdot a]_R \rangle \mid u \in \Sigma^*, a \in \Sigma\}$ we have $L(\mathcal{A}_{R,L}) = L$.

Proof. We first prove that δ_R is a correctly defined function - we have to show that the definition of function values does not depend on the choice of the member of the equivalence class. Let $v \in [u]_R$, i.e. $u \; R \; v$. Then, since R is right invariant we have $u \cdot a \; R \; v \cdot a$ for any $a \in \Sigma$. Hence

$$\delta_R([u]_R, a) = [u \cdot a]_R = [v \cdot a]_R = \delta_R([v]_R, a).$$

In a similar way we see that compatibility of L with R ensures that the definition of final states does not depend on the representative s chosen: if $u \in [s]_R$ and $s \in L$, then $u \in L$.
We finally show that $L = L(\mathcal{A}_{R,L})$. Let $u = a_1 a_2 \ldots a_k$. Consider the path

$$[\varepsilon] \rightarrow^{a_1} [a_1] \rightarrow^{a_2} [a_1 a_2] \rightarrow^{a_3} \ldots \rightarrow^{a_k} [u].$$

We have $u \in L(\mathcal{A}_{R,L})$ iff $[u]$ is a final state iff $u \in L$. □

The automaton $\mathcal{A}_{R,L}$ introduced in Proposition 3.4.4 is called the *canonical deterministic automaton for R and L*.

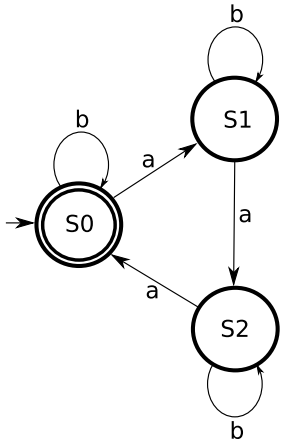

Figure 3.4: Canonical deterministic automaton for the equivalence relation $\sim$ and language L given in Example 3.4.3, cf. Example 3.4.5.

Example 3.4.5 Let $\sim$ and L be defined as in Example 3.4.3. Let $n_a(w)$ denote the number of occurrences of the letter a in w. The canonical deterministic automaton for $\sim$ and L has three states

$$\begin{aligned} S_0 &:= \{w \in \Sigma^* \mid n_a(w) \equiv 0 \ mod\ 3\} \\ S_1 &:= \{w \in \Sigma^* \mid n_a(w) \equiv 1 \ mod\ 3\} \\ S_2 &:= \{w \in \Sigma^* \mid n_a(w) \equiv 2 \ mod\ 3\} \end{aligned}$$

given by the equivalence classes of $\sim$. The final state, which corresponds to L, is S_0. The automaton is shown in Figure 3.4.

Remark 3.4.6 For the rest of this section, to simplify the formulation we make the following **general assumption**: *all automata considered are classical deterministic finite-state automata with total transition function, and each state is reachable from the start.*

Since we are interested in minimal deterministic automata, the accessibility condition helps to disregard automata that cannot be minimal for obvious reasons. In Remark 3.4.19 below we come back to the general situation where transition functions may be partial.

We now show that (under the general assumption) for each deterministic finite-state automaton $\mathcal{A}$ there exists an isomorphic automaton where the states are given as the equivalence classes of a relation $R_{\mathcal{A}}$ derived from $\mathcal{A}$.

Proposition 3.4.7 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic classical finite-state automaton. Then*

1. *$R_{\mathcal{A}} := \{\langle u, v \rangle \in \Sigma^* \times \Sigma^* \mid \delta^*(q_0, u) = \delta^*(q_0, v)\}$ is a right invariant equivalence relation and $L(\mathcal{A})$ is compatible with $R_{\mathcal{A}}$,*
2. *the automaton $\mathcal{A}_{R_{\mathcal{A}}, L(\mathcal{A})}$ is isomorphic to $\mathcal{A}$ by the state renaming function $h : \{[s]_{R_{\mathcal{A}}} \mid s \in \Sigma^*\} \to Q$ defined as $h([w]_{R_{\mathcal{A}}}) = \delta^*(q_0, w)$.*

Proof. 1. Let $u\ R_{\mathcal{A}}\ v$ and $w \in \Sigma^*$. Then

$$\delta^*(q_0, u \cdot w) = \delta^*(\delta^*(q_0, u), w) = \delta^*(\delta^*(q_0, v), w) = \delta^*(q_0, v \cdot w).$$

Hence $u \cdot w\ R_{\mathcal{A}}\ v \cdot w$, which shows that $R_{\mathcal{A}}$ is right invariant. If $u \in L(\mathcal{A})$ and $u\ R_{\mathcal{A}}\ v$, then $\delta^*(q_0, v) = \delta^*(q_0, u) \in F$ and thus $v \in L(\mathcal{A})$.
2. Since δ is total, $\delta^*(q_0, w)$ is defined for all $w \in \Sigma^*$. Clearly, for any $w \in \Sigma^*$ the state $\delta^*(q_0, w)$ determines the equivalence class of w and therefore h is an injective function. Since all states in Q are reachable the mapping h is surjective. Hence h is a bijection. Obviously $h([\varepsilon]) = q_0$ and $h(\{[s]_{R_{\mathcal{A}}} \mid s \in L(\mathcal{A})\}) = F$. The definitions of $R_{\mathcal{A}}$ and the transition function in $\mathcal{A}_{R_{\mathcal{A}}, L(\mathcal{A})}$ show that for every $u \in \Sigma^*$ and $a \in \Sigma$ we have $\delta(h([u]_{R_{\mathcal{A}}}), a) = h([ua]_{R_{\mathcal{A}}})$. It follows that $\mathcal{A}_{R_{\mathcal{A}}, L(\mathcal{A})}$ is isomorphic to $\mathcal{A}$ by the state renaming function h. □

In what follows, $R_{\mathcal{A}}$ is called the *equivalence relation induced by* $\mathcal{A}$, and $\mathcal{A}_{R_{\mathcal{A}}, L(\mathcal{A})}$ is called the *canonical automaton isomorphic to* $\mathcal{A}$. Note that number of states of $\mathcal{A}$ and $\mathcal{A}_{R_{\mathcal{A}}, L(\mathcal{A})}$ coincides with the index $|\Sigma^*/R_{\mathcal{A}}|$ of $R_{\mathcal{A}}$.

Definition 3.4.8 Let $L \subseteq \Sigma^*$ be a language over Σ. Then the relation

$$R_L = \{\langle u, v\rangle \in \Sigma^* \times \Sigma^* \mid \forall w \in \Sigma^* : u \cdot w \in L \text{ iff } v \cdot w \in L\}$$

is called the *Myhill-Nerode relation* for the language L.

Proposition 3.4.9 *Let $L \subseteq \Sigma^*$ be a language over Σ. Then the Myhill-Nerode relation R_L is a right invariant equivalence relation and R_L is compatible with L.*

Proof. Let $u\ R_L\ v$ and $w \in \Sigma^*$. In order to prove that $u \cdot w\ R_L\ v \cdot w$ we show that for any $w' \in \Sigma^*$ we have $(u \cdot w) \cdot w' \in L$ iff $(v \cdot w) \cdot w' \in L$. This is true since $u \cdot w'' \in L \leftrightarrow v \cdot w'' \in L$ holds for $w'' = w \cdot w'$ by definition of R_L. If $u \in L$ and $u\ R_L\ v$ then, by the definition of R_L, we have $v \in L$. □

Definition 3.4.10 Let R_L have finite index. Then the canonical deterministic automaton $\mathcal{A}_{R_L, L}$ for R_L and L is called the *Myhill-Nerode automaton* for the language L.

Note that the number of states of $\mathcal{A}_{R_L, L}$ coincides with the index $|\Sigma^*/R_L|$.

Example 3.4.11 Let $\sim$ and L be defined as in Examples 3.4.3 and 3.4.5. Then $\sim$ is the Myhill-Nerode relation for L, and the automaton shown in Figure 3.4 is the Myhill-Nerode automaton for L.

The following proposition shows that the equivalence relation induced by a deterministic finite-state automaton $\mathcal{A}$ is a refinement of the Myhill-Nerode relation of the language recognized by the automaton. This observation gives the key to later compare the size of $\mathcal{A}$ with the size of the Myhill-Nerode automaton for the automaton language.

Proposition 3.4.12 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta\rangle$ be a classical deterministic finite-state automaton. Then $R_{\mathcal{A}} \subseteq R_{L(\mathcal{A})}$.*

Proof. Let $u\ R_{\mathcal{A}}\ v$. Then $\delta^*(q_0,u)=\delta^*(q_0,v)$. Then for any $w\in\Sigma^*$ we have

$$\begin{aligned} u\cdot w\in L(\mathcal{A}) &\Leftrightarrow \delta^*(q_0,u\cdot w)\in F\\ &\Leftrightarrow \delta^*(\delta^*(q_0,u),w)\in F\\ &\Leftrightarrow \delta^*(\delta^*(q_0,v),w)\in F\\ &\Leftrightarrow \delta^*(q_0,v\cdot w)\in F\\ &\Leftrightarrow v\cdot w\in L(\mathcal{A}). \end{aligned}$$

Hence $u\ R_{L(\mathcal{A})}\ v$. □

Following the above general assumption, the minimality condition in the following theorem refers to the class of deterministic finite-state automata with total transition function.

Theorem 3.4.13 *For each classical deterministic finite-state automaton there exists a unique (up to renaming of states) equivalent deterministic finite-state automaton that is minimal with respect to the number of states.*

Proof. Let $\mathcal{A}=\langle\Sigma,Q,q_0,F,\delta\rangle$ be a classical deterministic finite-state automaton. We have seen that $|Q|=|\Sigma^*/R_{\mathcal{A}}|$. Proposition 3.4.12 shows that the number of states $|\Sigma^*/R_L|$ of the Myhill-Nerode automaton $\mathcal{A}_{R_L,L}$ for the language $L=L(\mathcal{A})$ does not exceed $|Q|$. Hence the number of states of the Myhill-Nerode automaton $\mathcal{A}_{R_L,L}$, which recognizes $L(\mathcal{A})$ (Proposition 3.4.4), is minimal among all deterministic finite-state automata equivalent to $\mathcal{A}$. Let $\mathcal{A}'$ be a deterministic finite-state automaton equivalent to $\mathcal{A}$ such that $\mathcal{A}'$ and $\mathcal{A}_{R_L,L}$ have the same number of states. Proposition 3.4.12 shows that $R_{\mathcal{A}'}=R_{L(\mathcal{A})}$. Proposition 3.4.7 shows that $\mathcal{A}_{R_{\mathcal{A}'},L(\mathcal{A}')}=\mathcal{A}_{R_L,L}$ is isomorphic to $\mathcal{A}'$. □

Theorem 3.4.14 *Let $L\subseteq\Sigma^*$ be a classical language. Then L is a classical automaton language iff the index of R_L is finite.*

Proof. If L is a classical automaton language, let $\mathcal{A}$ be a deterministic finite-state automaton recognizing L with set of states Q. As in the above proof we see that $|\Sigma^*/R_L|$ does not exceed $|Q|$. Conversely, if $|\Sigma^*/R_L|$ is finite, the Myhill-Nerode automaton for L recognizes L. □

Combining Theorem 3.4.14 and Theorem 2.4.2 we obtain

Theorem 3.4.15 *Let $L\subseteq\Sigma^*$ be a classical language. Then the following are equivalent:*

1. *L is a classical automaton language,*
2. *L is a regular language,*
3. *The index of R_L is finite.*

Remark 3.4.16 Theorem 3.4.15 can be used to directly prove that specific languages $L\subseteq\Sigma^*$ are regular/non-regular. As a first example, consider the language

$$L=\{a^nb^n\mid n\in\mathbb{N}\}.$$

Obviously all strings in the set $\{a^n \mid n \in \mathbb{N}\}$ are non-equivalent with respect to the Myhill-Nerode relation R_L. Hence L is not regular. As a second example, let Σ be an arbitrary alphabet containing the letter a. Let E (O) denote the set of words over Σ with an even (odd) number of occurrences of the letter a. Then all elements in E and O are equivalent with respect to the relations R_E and R_O, which shows that E and O are regular languages.

Minimal deterministic finite-state automata have characteristic structural properties. Following our general assumption, in the following proposition we consider automata with total transition function where each state is reachable from the start. The next proposition gives a characterization of the minimal automaton in terms of equivalent states (cf. Definition 2.1.18).

Proposition 3.4.17 *A deterministic finite-state automaton $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ is minimal iff distinct states of $\mathcal{A}$ are never equivalent.*

Proof. First note the following equivalences, which hold for all $u, v \in \Sigma^*$:

$$\begin{aligned} [u]_{R_{L(\mathcal{A})}} = [v]_{R_{L(\mathcal{A})}} &\Leftrightarrow \forall w \in \Sigma^* : uw \in L(\mathcal{A}) \text{ iff } vw \in L(\mathcal{A}) \\ &\Leftrightarrow \forall w \in \Sigma^* : \delta^*(q_0, uw) \in F \text{ iff } \delta^*(q_0, vw) \in F \\ &\Leftrightarrow \delta^*(q_0, u) \equiv \delta^*(q_0, v). \end{aligned}$$

We obtain the following equivalences. The second equivalence is a consequence of Proposition 3.4.12.

$$\begin{aligned} \mathcal{A} \text{ is not minimal} &\Leftrightarrow |R_{L(\mathcal{A})}/\Sigma^*| < |R_{\mathcal{A}}/\Sigma^*| \\ &\Leftrightarrow \exists u, v \in \Sigma^* : [u]_{R_{\mathcal{A}}} \neq [v]_{R_{\mathcal{A}}} \;\&\; [u]_{R_{L(\mathcal{A})}} = [v]_{R_{L(\mathcal{A})}} \\ &\Leftrightarrow \exists u, v \in \Sigma^* : \delta^*(q_0, u) \neq \delta^*(q_0, v) \;\&\; [u]_{R_{L(\mathcal{A})}} = [v]_{R_{L(\mathcal{A})}} \\ &\Leftrightarrow \exists u, v \in \Sigma^* : \delta^*(q_0, u) \neq \delta^*(q_0, v) \;\&\; \delta^*(q_0, u) \equiv \delta^*(q_0, v). \end{aligned}$$

□

Departing from the general assumption, for the sake of generality in the next proposition we consider trimmed deterministic finite-state automata, which in general have a partial transition function. For such automata an inductive characterization of the equivalence of two states (cf. Def. 2.1.18) can be given.

Proposition 3.4.18 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a trimmed classical deterministic finite-state automaton. Then q_1 and q_2 are equivalent iff*

1. *$q_1 \in F$ iff $q_2 \in F$,*
2. *for all $\sigma \in \Sigma$ the following holds: $\delta(q_1, \sigma)$ is defined iff $\delta(q_2, \sigma)$ is defined. If both are defined, then $\delta(q_1, \sigma)$ and $\delta(q_2, \sigma)$ are equivalent.*

Note that this characterization is an immediate consequence of Proposition 2.1.17.

Remark 3.4.19 The Myhill-Nerode automaton for a classical regular language L as defined above may contain a unique state q from which we cannot reach a final state. In this case, when we are interested in minimal deterministic automata with *partial* transition function for L, then this state and all transitions leading to q have to be deleted. Cf. Exercise 3.9.6.

3.5 Minimization of deterministic finite-state automata

Given a deterministic finite-state automaton $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ for the language $L(\mathcal{A}) = L$ we now show how to build an equivalent minimal automaton $\mathcal{A}_L$ by simultaneously identifying all equivalent states. **For simplicity we assume that the transition function δ of $\mathcal{A}$ is total.** Consider Definition 2.1.18 and Remark 3.4.17. To construct $\mathcal{A}_L$ we have to build the classes of equivalent states of $\mathcal{A}$. For all $p, q \in Q$ we have

$$\begin{aligned} q \equiv p &\Leftrightarrow L_{\mathcal{A}}(q) = L_{\mathcal{A}}(p) \\ &\Leftrightarrow \forall \alpha \in \Sigma^* : (\delta^*(q, \alpha) \in F \leftrightarrow \delta^*(p, \alpha) \in F). \end{aligned}$$

In general it is difficult to directly find all classes of equivalent states. Using a more indirect procedure we now introduce a sequence of equivalence relations

$$R_0, R_1, R_2, \ldots$$

on Q. We shall see that

1. we may compute R_0 directly, and R_{i+1} can be computed from R_i for all $i \geq 0$.
2. after at most $|Q| - 1$ steps the sequence becomes stationary, and the final equivalence relation obtained is the desired equivalence relation $\equiv$.

Definition 3.5.1 The relations $R_i \subseteq Q \times Q$ $(i \geq 0)$ are formally defined as

$$q \; R_i \; p \;:\Leftrightarrow\; \forall \alpha \in \Sigma^* \text{ s.th. } |\alpha| \leq i : (\delta^*(q, \alpha) \in F \leftrightarrow \delta^*(p, \alpha) \in F).$$

Note that this condition relaxes the above condition for $p \equiv q$, only looking at words α of length $\leq i$. Clearly R_{i+1} is always a refinement of R_i $(i \geq 0)$ and we have $q \equiv p$ iff $q \; R_i \; p$ for all $i \geq 0$. For the direct computation of R_0 we use that

$$q \; R_0 \; p \Leftrightarrow (q \in F \leftrightarrow p \in F).$$

The next lemma shows how R_{i+1} can be obtained from R_i.

Lemma 3.5.2 *For all states $q, p \in Q$ the following two conditions are equivalent:*

1. $\forall \alpha \in \Sigma^*$ *s.th.* $|\alpha| \leq i + 1 : (\delta^*(q, \alpha) \in F \leftrightarrow \delta^*(p, \alpha) \in F)$,
2. *(a)* $\forall \alpha \in \Sigma^*$ *s.th.* $|\alpha| \leq i : (\delta^*(q, \alpha) \in F \leftrightarrow \delta^*(p, \alpha) \in F)$, *and*
   (b) $\forall \sigma \in \Sigma, \forall \alpha \in \Sigma^*$ *s.th.* $|\alpha| \leq i : (\delta^*(\delta(q, \sigma), \alpha) \in F \leftrightarrow \delta^*(\delta(p, \sigma), \alpha) \in F)$.

The following proposition follows from the above equivalence.

Proposition 3.5.3

$$q \; R_{i+1} \; p \;\Leftrightarrow\; q \; R_i \; p \;\&\; \forall a \in \Sigma : \delta(q, a) \; R_i \; \delta(p, a)$$

This shows how R_{i+1} can be computed from R_i. We also see that once $R_{i+1} = R_i$ we also have $R_{i+k} = R_i$ for all $k \geq 0$. Since R_{i+1} is always a refinement of R_i and we cannot have more than $|Q|$ equivalence classes it follows that the sequence $R_0, R_1, \ldots$ becomes stationary, say at R_k. Since $q \equiv p$ iff $q\ R_i\ p$ for all $i \geq 0$ it follows that R_k coincides with $\equiv$.

Corollary 3.5.4 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic finite-state automaton where δ is total and each state is reachable. Then the relation $R = \bigcap_{i=0}^{\infty} R_i$, where*

$$\begin{array}{rcl} q\ R_0\ p & \Leftrightarrow & (q \in F \leftrightarrow p \in F) \\ q\ R_{i+1}\ p & \Leftrightarrow & q\ R_i\ p\ \&\ \forall a \in \Sigma : \delta(q,a)\ R_i\ \delta(p,a) \end{array}$$

coincides with the equivalence of states $\equiv$ for $\mathcal{A}$. The automaton

$$\mathcal{A}' = \langle \Sigma, \{[q]_R \mid q \in Q\}, [q_0]_R, \{[f]_R \mid f \in F\}, \delta' \rangle$$

where $\delta'([q]_R), \sigma) := [\delta(q,\sigma)]_R$ is the minimal deterministic automaton equivalent to $\mathcal{A}$.

We now look at algorithmic details. When we want to compute the relations R_{i+1}, given R_i, we need a procedure for testing the conditions on the right-hand side of the second equivalence above. Instead of looking at each candidate pair $\langle p, q \rangle$ in isolation we shall find the full relation R_{i+1} directly by computing the intersection of R_i with several equivalence relations. We denote the intersection of two equivalence relations $S, T \subseteq Q \times Q$ as $S \cap T$. The following notions lead to the equivalence relations needed.

Definition 3.5.5 Let $g : Q \to X$. Then the equivalence relation $ker_Q(g) \subseteq Q \times Q$ defined as

$$\langle p, q \rangle \in ker_Q(g) \ :\Leftrightarrow\ g(p) = g(q)$$

is called the *kernel* of g over Q.

Proposition 3.5.6 *Let us define $f : Q \to \{0, 1\}$*

$$f(q) := \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases}$$

and for every $a \in \Sigma$ and $i \in \mathbb{N}$ the function $f_a^{(i)} : Q \to Q/R_i$ as

$$f_a^{(i)}(q) := [\delta(q,a)]_{R_i}.$$

Then

1. $R_0 = ker_Q(f)$
2. $R_{i+1} = \bigcap_{a \in \Sigma} ker_Q(f_a^{(i)}) \cap R_i$

The second equation expresses that R_{i+1} is obtained from R_i by restricting the relation to pairs q and q such that for all $\sigma \in \Sigma$ the σ-successors of p and q have the same equivalence class with respect to R_i. Proposition 3.5.6 represents our final reformulation of the equivalences in Corollary 3.5.4 used for programs.

Example 3.5.7 Figure 3.5 illustrates the inductive computation of the relation $\equiv$ for a deterministic finite-state automaton $\mathcal{A}$ and the Myhill-Nerode automaton for $L(\mathcal{A})$. The example automaton with 10 states accepts the two words *aaaa* and *baaa*. In the upper representation of the automaton, the two equivalence classes of R_0 (final states, non-final states) are shown using two boxes. From each of the two states 4 and 8, with letter a we reach a final state, while with letter b we reach a non-final state. Hence at the next step, States 4 and 8 remain equivalent. In contrast, from states 1, 2, 3, 6, 7, and 10, both transitions lead to a non-final state. For R_1 we obtain the three equivalence classes $\{5,9\},\{4,8\},\{1,2,3,4,7,10\}$ shown in the second diagram. Continuing in the same way, in this example we obtain four equivalence classes for R_2, five equivalence classes for R_3 and six equivalence classes for R_4. Note that each relation R_{i+1} represents a refinement of R_i. The construction stops after reaching R_4 since no further refinement is obtained, we have $R_4 = R_5 = R_6 = \ldots$.

Remark 3.5.8 Recall that the above definitions and algorithmic descriptions assume that the transition function of input automata is total. However, in many practical situations deterministic automata are found that - for the sake of space economy - come with a partially defined transition function. There are several ways of generalizing the above construction to this more general case.

We first consider *trimmed* deterministic input automata. Note that all transitions of a trimmed automton are "promising" in the sense that we may reach a final state using the transition. There are no "useless" states. For trimmed deterministic input automata with partial transition function, the above inductive definition of R_{i+1} in Corollary 3.5.4 is modified, defining

$$q\ R_{i+1}\ p \Leftrightarrow q\ R_i\ p\ \&\ \forall a \in \Sigma : \delta(q,a), \delta(p,a) \text{ undefined or } \delta(q,a)\ R_i\ \delta(p,a).$$

In Proposition 3.5.6 the definition of the functions $f_a^{(i)}$ has to be adapted accordingly, introducing a special value $\perp$ for $f_a^{(i)}(q)$ if $\delta(q,a)$ is undefined. In Corollary 3.5.4 the automaton $\mathcal{A}' = \langle \Sigma, \{[q]_R \mid q \in Q\}, [q_0]_R, \{[f]_R \mid f \in F\}, \delta' \rangle$ is now defined with a partial transition function: $\delta'([q]_R), \sigma)$ is defined (as $[\delta(q,\sigma)]_R$) iff $\delta(q,\sigma)$ is defined. With these modifications, $\mathcal{A}'$ is the minimal deterministic automaton with partial transition function equivalent to $\mathcal{A}$.

A straightforward, but more space-consuming method can be used for arbitrary deterministic input automata $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$: if δ is not total, just introduce one new nonfinal state b and minimize the variant input automaton $\mathcal{A}' = (\Sigma, Q \cup \{b\}, q_0, F, \delta')$ where δ' is the extension of δ to a total transition function such that

- $\delta'(q,\sigma) := b$ whenever $\delta(q,\sigma)$ is undefined $(q \in Q, \sigma \in \Sigma)$,
- $\delta'(b,\sigma) := b$ for all $\sigma \in \Sigma$.

Obviously, $\mathcal{A}$ and $\mathcal{A}'$ are equivalent. The minimal deterministic automaton obtained when minimizing $\mathcal{A}'$ using the above standard construction contains exactly one state that can be deleted when considering minimal deterministic automata with partial transition function, cf. Exercise 3.9.6. The disadvantage of this method relies on the fact that the number of new transitions introduced in $\mathcal{A}'$ can be large.

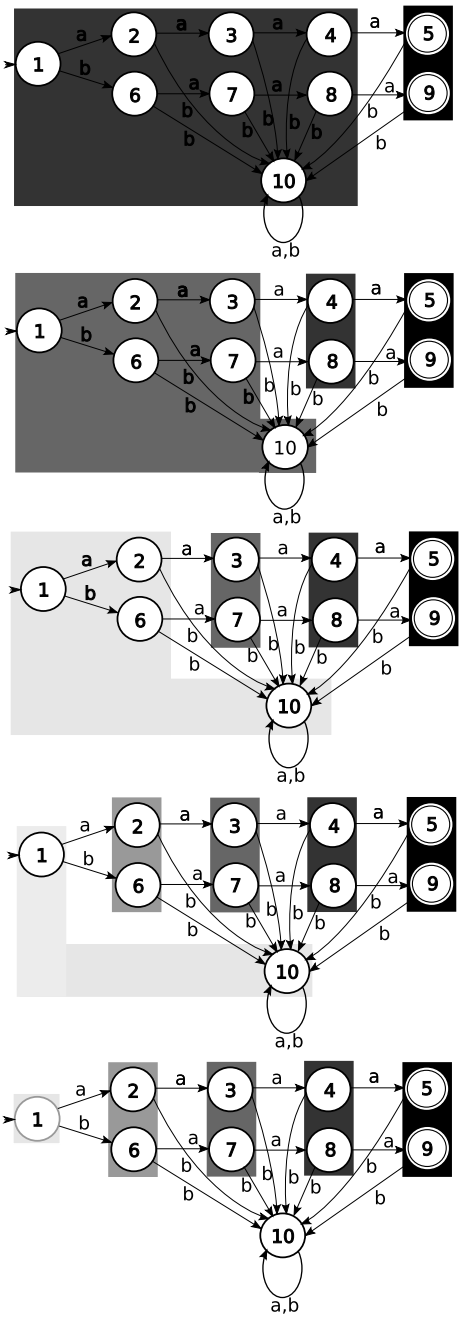

Figure 3.5: Inductive minimization of a deterministic finite-state automaton, cf. Example 3.5.7. At each step, equivalence classes are split looking for each state at the classes reached with letters a and b. The minimal automaton has 6 states corresponding to the equivalence classes shown in the bottom diagram.

3.6 Coloured deterministic finite-state automata

In many cases it is useful to categorize the words recognized by an automaton into a finite set of classes. Following this view we introduce a simple generalization of deterministic finite-states automata called coloured deterministic finite-state automata and show how this generalization can be represented by classical automata. A generalization of the above minimization technique for the case of coloured automata is presented. For simplicity we only look at deterministic automata with totally defined transition function.

An abstract view of formalizing properties of states is captured by the following definition.

Definition 3.6.1 Let C be a finite set called the *set of colours*. A *C-coloured deterministic finite-state automaton* is a deterministic finite-state automaton $\langle \Sigma, Q, q_0, F, \delta \rangle$ with total transition function δ together with a surjective total function $col : F \to C$. We write $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ for the coloured automaton. Colour $col(q)$ is called the colour of state $q \in F$. The *language $L(\mathcal{A})$ accepted by a coloured automaton $\mathcal{A}$* is defined as usual, ignoring colours.

Throughout this section we assume that a finite set of colours C is given and that $C \cap \Sigma = \emptyset$.

Definition 3.6.2 Let $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ be a C-coloured deterministic finite-state automaton. The *colouring of words recognized by $\mathcal{A}$* is the function $col_{\mathcal{A}} : L(\mathcal{A}) \to C, w \mapsto col(\delta^*(q_0, w))$.

Proposition 3.6.3 *Let $L \subseteq \Sigma^*$ be a language and let $c : L \to C$ be a colouring function for the words of L. Consider the modified language*

$$L_c := \{\alpha \cdot c(\alpha) \mid \alpha \in L\} \subseteq \Sigma^* \cdot C.$$

Let $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ be a C-coloured deterministic finite-state automaton such that $L(\mathcal{A}) = L$ and $col_{\mathcal{A}} = c$. Let

$$\mathcal{A}_c := \langle \Sigma \cup C, Q \cup \{f\}, q_0, \{f\}, \delta \cup \{\langle q, col(q), f \rangle \mid q \in F\} \rangle$$

where $f \notin Q$ is a new state. Then $L(\mathcal{A}_c) = L_c$.

Using the correspondence from Proposition 3.6.3 it is possible to transfer classical automata results to the case of coloured automata, including minimization results. In order to avoid the above translation we now explicitly describe minimization of coloured automata.

Minimization of coloured deterministic finite-state automata

Proofs of the following results are directly obtained from the corresponding proofs for the case of classical automata.

Definition 3.6.4 Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be C-coloured deterministic finite-state automata. $\mathcal{A}_1$ and $\mathcal{A}_2$ are *equivalent* iff $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ and $col_{\mathcal{A}_1} = col_{\mathcal{A}_2}$.

Definition 3.6.5 A language $L \subseteq \Sigma^*$ with a colouring $c : L \to C$ is *compatible* with the equivalence relation $R \subseteq \Sigma^* \times \Sigma^*$ iff

$$\forall u, v \in \Sigma^* : u \in L \;\&\; u \; R \; v \;\rightarrow\; (v \in L \;\&\; c(u) = c(v)).$$

Proposition 3.6.6 *Let $R \subseteq \Sigma^* \times \Sigma^*$ be a right invariant equivalence relation such that the index of R is finite, let $L \subseteq \Sigma^*$ be a language over Σ with a colouring $c : L \to C$ compatible with R. Then for the coloured deterministic finite-state automaton*

$$\mathcal{A}_{R,L,c} := \langle \Sigma, C, \{[s]_R \mid s \in \Sigma^*\}, [\varepsilon]_R, \{[s]_R \mid s \in L\}, \delta, col \rangle$$

with transition function $\delta := \{\langle [u]_R, a, [u \cdot a]_R \rangle \mid u \in \Sigma^*, a \in \Sigma\}$ and colouring $col = \{\langle [s]_R, c(s) \rangle \mid s \in L\}$ we have $L(\mathcal{A}_{R,L,c}) = L$ and $col_{\mathcal{A}_{R,L,c}} = c$.

Definition 3.6.7 Let

$$\begin{aligned} \mathcal{A}' &= \langle \Sigma, C, Q', q_0', F', \delta', col' \rangle \\ \mathcal{A}'' &= \langle \Sigma, C, Q'', q_0'', F'', \delta'', col'' \rangle \end{aligned}$$

be two coloured deterministic finite-state automata. $\mathcal{A}'$ is *isomorphic* to $\mathcal{A}''$ by the *state renaming function* $f : Q' \to Q''$ if f is a bijection such that

- $f(q_0') = q_0''$,
- $f(F') = F''$,
- for all $q' \in Q'$ and all $a \in \Sigma$ we have $f(\delta'(q', a)) = \delta''(f(q'), a)$, and
- for all $q' \in F'$ we have $col'(q') = col''(f(q'))$.

Proposition 3.6.8 *Let $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ be a coloured deterministic finite-state automaton. Then*

1. *$R_{\mathcal{A}} := \{\langle u, v \rangle \in \Sigma^* \times \Sigma^* \mid \delta^*(q_0, u) = \delta^*(q_0, v)\}$ is a right invariant equivalence relation and $L(\mathcal{A})$ with colouring $col_{\mathcal{A}}$ is compatible with $R_{\mathcal{A}}$,*
2. *the automaton $\mathcal{A}_{R_{\mathcal{A}}, L(\mathcal{A}), col_{\mathcal{A}}}$ is isomorphic to $\mathcal{A}$ by the state renaming function $h : \{[s]_{R_{\mathcal{A}}} \mid s \in \Sigma^*\} \to Q$ defined as $h([w]_{R_{\mathcal{A}}}) := \delta^*(q_0, w)$.*

Definition 3.6.9 Let $L \subseteq \Sigma^*$ be a language and let $c : L \to C$ be a colouring of L. For $\langle u, v \rangle \in \Sigma^* \times \Sigma^*$ we define

$$u \; R_{L,c} \; v :\leftrightarrow \forall w \in \Sigma^* : (u \cdot w \in L \leftrightarrow v \cdot w \in L) \;\&\; (u \cdot w \in L \rightarrow c(u \cdot w) = c(v \cdot w)).$$

The relation $R_{L,c}$ is called the *Myhill-Nerode relation* for the pair $\langle L, C \rangle$.

Definition 3.6.10 Let $R_{L,c}$ have finite index. Then the automaton $\mathcal{A}_{R_{L,c}, L, c}$ is called the *Myhill-Nerode automaton* for the language L with colouring c.

Proposition 3.6.11 *Let $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ be a coloured deterministic finite-state automaton. Then $R_{\mathcal{A}} \subseteq R_{L(\mathcal{A}), col_{\mathcal{A}}}$.*

Theorem 3.6.12 *For each coloured deterministic finite-state automaton there exists a unique (up to renaming of states) equivalent coloured deterministic finite-state automaton that is minimal with respect to the number of states.*

Definition 3.6.13 Let $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ be a coloured deterministic finite-state automaton. Two states p, q of $\mathcal{A}$ are *equivalent* if $\forall w \in \Sigma^* : (\delta^*(p,w) \in F \leftrightarrow \delta^*(q,w) \in F) \,\&\, (\delta^*(p,w) \in F \rightarrow col(\delta^*(p,w)) = col(\delta^*(q,w)))$.

Proposition 3.6.14 *A coloured deterministic finite-state automaton $\mathcal{A}$ is minimal iff distinct states of $\mathcal{A}$ are never equivalent.*

The procedure for finding the minimal C-coloured deterministic finite-state automaton $\mathcal{A}'$ equivalent to a given coloured deterministic automaton $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ with total transition function where each state is reachable is similar as in the uncoloured case.

Definition 3.6.15 We introduce equivalence relations R_i $(i \geq 0)$ on Q, in this case inductively defining

$$\begin{aligned} q \; R_0 \; p &\quad\leftrightarrow\quad (q \in F \leftrightarrow p \in F) \,\&\, (q \in F \rightarrow col(q) = col(p)) \\ q \; R_{i+1} \; p &\quad\leftrightarrow\quad q \; R_i \; p \,\&\, \forall a \in \Sigma : \delta(q,a) \; R_i \; \delta(p,a). \end{aligned}$$

If $q \; R_i \; p$ and q' and p' respectively are states that are reached from q and p with the same word w of length $\leq i$, then $(q' \in F \leftrightarrow p' \in F) \,\&\, (q' \in F \rightarrow col(q') = col(p'))$. As in the uncoloured case we obtain the following corollary.

Corollary 3.6.16 *Let $\mathcal{A} = \langle \Sigma, C, Q, q_0, F, \delta, col \rangle$ be a coloured deterministic finite-state automaton where δ is total and each state is reachable, let R_i defined as above $(i \geq 0)$. Then $R = \bigcap_{i=0}^{\infty} R_i$ coincides with the equivalence of states $\equiv$ on $\mathcal{A}$. The coloured automaton*

$$\mathcal{A}' = \left\langle \Sigma, C, \{[q]_R \mid q \in Q\}, [q_0]_R, \{[f]_R \mid f \in F\}, \delta', col' \right\rangle$$

where $\delta'([q]_R), \sigma) := [\delta(q,\sigma)]_R$ and $col'([q]_R) := col(q)$ is the minimal coloured deterministic automaton equivalent to $\mathcal{A}$.

Proposition 3.6.17 *Let us define $f : Q \rightarrow C \cup \{0_c\}$, where $0_c \notin C$ is an arbitrary symbol*

$$f(q) := \begin{cases} c(q) & \text{if } q \in F \\ 0_c & \text{otherwise} \end{cases}$$

and for every $a \in \Sigma$ and $i \in \mathbb{N}$ the function $f_a^{(i)} : Q \rightarrow Q/R_i$ as

$$f_a^{(i)}(q) := [\delta(q,a)]_{R_i}.$$

Then

1. $R_0 = ker_Q(f)$
2. $R_{i+1} = \bigcap_{a \in \Sigma} ker_Q(f_a^{(i)}) \cap R_i$.

Example 3.6.18 Figure 3.6 illustrates the inductive computation of the relation $\equiv$ for a coloured deterministic finite-state automaton $\mathcal{A}$. The two final states have distinct colours (graphically indicated by using another background). In the upper representation of the automaton, the three equivalence classes of R_0 (two classes of final states, non-final states) are shown. From each of the two states 4 and 8, with letter a we reach a final state, while with letter b we reach a non-final state. However, in this case the final states reached have distinct colours. Hence at the next step, States 4 and 8 represent distinct equivalence classes. From states 1, 2, 3, 6, 7, and 10, both transitions lead to a non-final state. For R_1 we obtain the five equivalence classes $\{5\}, \{9\}, \{4\}, \{8\}, \{1, 2, 3, 4, 7, 10\}$ shown in the second diagram. Continuing in the same way, in this example we obtain seven equivalence classes for R_2, nine equivalence classes for R_3 and ten equivalence classes for R_4. The result shows that the input coloured automaton us already minimal.

3.7 Pseudo-determinization and pseudo-minimization of monoidal finite-state automata

It is natural to ask which of the additional closure results obtained for classical regular languages and classical finite-state automata in Section 3.3 can be lifted to the general monoidal case. A simple example (cf. [Kaplan and Kay, 1994]) shows that the set of monoidal regular languages is *not closed under intersection*: we consider the monoidal regular languages R_1, R_2 in the monoid $\langle \Sigma^* \times \Sigma^*, \cdot, \langle \varepsilon, \varepsilon \rangle \rangle$:

$$\begin{array}{ll} R_1 = \langle a, c \rangle^* \cdot \langle b, \varepsilon \rangle^* & R_1 = \{ \langle a^n b^m, c^n \rangle \mid n, m \in \mathbb{N} \} \\ R_2 = \langle a, \varepsilon \rangle^* \cdot \langle b, c \rangle^* & R_2 = \{ \langle a^m b^n, c^n \rangle \mid n, m \in \mathbb{N} \} \end{array}$$

The intersection of R_1 and R_2 is

$$R_1 \cap R_2 = \{ \langle a^n b^n, c^n \rangle \mid n \in \mathbb{N} \}$$

which is not regular since the first projection is not a regular language (cf. Remark 3.4.16) – in the next chapter we show that monoidal regular languages are closed under projection, see Proposition 4.2.1. From the above property it follows directly that the set of monoidal regular languages is *not closed under complement and difference*. Otherwise using the de Morgan laws the monoidal regular languages would be closed under intersection.

Looking at the proof of Proposition 3.3.1 we see that the closure of classical regular languages an over alphabet Σ under complement is derived from the following observation. We may represent the language using a deterministic automaton with total transition function. Then for every word $w \in \Sigma^*$ there exists a unique path from the start state with label w. Since the monoidal regular languages are not closed under complement it is clear that there is no construction which ensures this "unique path" property for monoidal automata. This shows that the definition of a deterministic monoidal finite-state automaton is not natural. In the following chapters we shall discuss specialized notions of determinism for some classes of monoidal finite-state automata. We now introduce

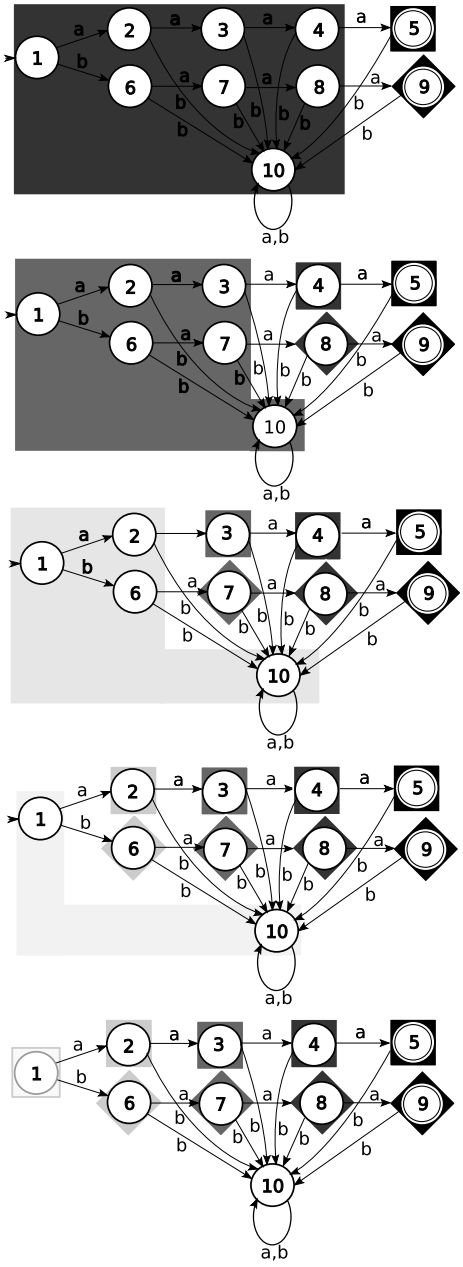

Figure 3.6: Inductive minimization of a coloured deterministic finite-state automaton, cf. Example 3.6.18. The minimal coloured automaton has 10 states corresponding to the equivalence classes shown in the bottom diagram. Here it is isomorphic to the input coloured automaton.

a weaker notion, using the fact that each monoidal finite-state automaton can be represented as the homomorphic image of a classical finite-state automaton (cf. Theorem 2.1.22).

Definition 3.7.1 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. A monoidal finite-state automaton $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ is called *pseudo-deterministic* if there exists exactly one initial state in I and for any state $q \in Q$ and any $m \in M$ there exists at most one state q' such that $\langle q, m, q'\rangle \in \Delta$.

The definition expresses that the free companion (cf. Definition 2.1.23) of $\mathcal{A}$ is deterministic. [1]

Proposition 3.7.2 *For each monoidal finite-state automaton $\mathcal{A}$ we may effectively construct a pseudo-deterministic monoidal finite-state automaton $\mathcal{A}'$ equivalent to $\mathcal{A}$.*

Proof. Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ be as above. Let

$$\mathcal{A}' = \langle \Sigma, Q, I, F, \Delta\rangle$$

denote the free companion of $\mathcal{A}$. Recall that the letters in Σ have the form a_m where m is a transition label of $\mathcal{A}$. Following Theorem 3.2.2, let

$$\mathcal{A}'' = \langle \Sigma, Q', q_0, F', \delta\rangle$$

denote a deterministic finite-state automaton equivalent to $\mathcal{A}'$. Let

$$\mathcal{A}''' = \langle \mathcal{M}, Q', q_0, F', \Delta'\rangle$$

denote the image of $\mathcal{A}''$ under the homomorphism h induced by the mapping $a_m \mapsto m$ (cf. proof of Theorem 2.1.22). Then $\mathcal{A}'''$ is pseudo-deterministic and we have $L(\mathcal{A}''') = h(L(\mathcal{A}'')) = h(L(\mathcal{A}')) = L(\mathcal{A})$. □

Definition 3.7.3 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. A pseudo-deterministic monoidal finite-state automaton $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ is called *pseudo-minimal* if the free companion of $\mathcal{A}$ is a minimal deterministic finite-state automaton.

Proposition 3.7.4 *For each monoidal finite-state automaton $\mathcal{A}$ we may effectively construct a pseudo-minimal monoidal finite-state automaton $\mathcal{A}'$ equivalent to $\mathcal{A}$.*

Proof. The proof is an obvious variant of the proof of Proposition 3.7.2, applying a minimization step for the deterministic classical finite-state automaton built during the construction. □

[1] Sometimes a more restrictive definition is used, demanding in addition that $\mathcal{A}$ is e-free.

	Monoidal view	Classical (string) view
Device	Monoidal finite-state automaton	Classical finite-state automaton
Languages recognized	Monoidal automaton language	Classical automaton language
Closed under	Union, concatenation, Kleene-Star Homomorphic images	Union, concatenation, Kleene-Star Homomorphic images, intersection, Complement, difference Reverse language
Equivalent lang. concept	Monoidal regular language	Classical regular language
Expressions	Monoidal regular expression	Classical regular expression
Operations for automata	Removal of ϵ-transitions Pseudo-determinization Pseudo-minimization	Removal of ϵ-transitions Determinization Minimization

Table 3.1: Finite-state automata and regular languages from a monoidal and a classical (string-based) view.

3.8 Summing up

In this chapter we studied *classical* finite-state automata, a special case of monoidal finite-state automata. Generalizing the closure properties for monoidal finite-state automata obtained in the previous chapter we showed that classical finite-state automata are in addition closed under intersection, difference, complement, and reversion of languages (Proposition 3.3.5). Furthermore, for classical finite-state automata there exists a determinization procedure (Theorem 3.2.2). We introduced the Myhill-Nerode equivalence relation, which gives an algebraic view on regular languages and deterministic finite-state automata. Using this picture, for deterministic finite-state automata a minimization procedure was given, which was extended to the case of coloured deterministic finite-state automata. Table 3.1 compares the general monoidal view with the classical (string based) view.

3.9 Exercises for Chapter 3

Exercise 3.9.1 Let Σ be a finite alphabet. Define a non-deterministic automaton with input alphabet Σ and set of states of the form $Q = \{s\} \cup \{q_\sigma \mid \sigma \in \Sigma\}$ such that the set of active states for each non-empty input w is $\{q_\sigma \mid \sigma \text{ occurs in } w\}$. Show that the number of states of the deterministic automaton reached by the optimized determinization procedure described in Remark 3.2.3 is at least $2^{|\Sigma|}$.

Exercise 3.9.2 Let w be a non-empty string over the alphabet Σ.

1. Let L_1 denote the set of strings over Σ that contain exactly one occurrence of w as infix. Show that L_1 is regular. (Note that in general different occurrences of w in a string v can overlap!).

2. Let L_2 denote the set of strings over Σ that contain exactly two occurrences of w as infix. Show that L_2 is regular.

Exercise 3.9.3 Assume we are given

1. a finite set P of pre-names (such as “Bob”),

2. a finite set F of family names (such as “Miller”),

3. a finite set T of City names (such as "New York").

How can we build a deterministic finite-state automaton $\mathcal{A}$ such that $\mathcal{A}$ accepts a string w iff the string has an infix that represents an "address" of the form "Bob Miller, New York"? Formalize the notion of an address and describe options.

Exercise 3.9.4 The symmetric difference of two sets M, N is $(M \setminus N) \cup (N \setminus M)$. Let $L_1, L_2 \subseteq \Sigma^*$ be two classical automaton languages. Use the Cartesian product of automata to define a classical finite-state automaton $\mathcal{A}$ such that $L(\mathcal{A})$ is the symmetric difference of L_1 and L_2. Which assumptions are appropriate for the input automata $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively accepting L_1 and L_2?

Exercise 3.9.5 Let $\Sigma := \{a, b, c\}$. Describe the Myhill-Nerode Automaton for each of the following languages:

1. $L_1 :=$ set of all words over Σ with at most 2 occurrences of the letter a,
2. $L_2 :=$ set of all words w over Σ where w contains an even number of occurrences of a and an even number of occurrences of b,
3. $L_3 :=$ set of all words where the total number of occurrences of a or b is even.

Exercise 3.9.6 Let us call a classical language $L \subseteq \Sigma^*$ *dense* iff for each $u \in \Sigma^*$ there exists a word $w \in \Sigma^*$ such that $uw \in L$. Let L be a classical regular language. Show that the following properties are equivalent:

1. L is non-dense.
2. The Myhill-Nerode automaton for L contains exactly one state q from which no final state can be reached.

Exercise 3.9.7 Let $\Sigma := \{a, b\}$. Let $\sharp a(w)$ and $\sharp b(w)$ denote the number of occurrences of the letter a (b) in $w \in \Sigma^*$. Use Theorem 3.4.15 to show that the following languages are not regular:

1. $L_1 := \{w \in \Sigma^* \mid \sharp a(w) = \sharp b(w)\}$;
2. $L_1 := \{w \in \Sigma^* \mid \sharp a(w) \geq \sharp b(w)\}$;

Exercise 3.9.8 Let $\Sigma := \{a, b, c, d\}$. Let L denote the set of all words over Σ that have length ≥ 2 and begin and end with the same letter. Show that L is a regular language and build the minimal deterministic automaton accepting L.

Exercise 3.9.9 Let $\Sigma = \{a, b\}$ and $L = \Sigma^* \cdot \{a^2b\} \cup \{aba\} \cdot \Sigma^*$. Construct the minimimal deterministic automaton accepting L.

Exercise 3.9.10 For each $n > 1$ give a classical deterministic finite-state automaton $\mathcal{A}_n$ with exactly $n + 1$ states such that the minimization procedure described in Section 3.5 needs $n - 1$ steps.

Exercise 3.9.11 Consider the deterministic finite-state automaton

$$A = (\{a, b\}, \{1, 2, 3, 4, 5, 6, 7, 8, 9\}, 1, \{2, 4, 9\}, \delta)$$

where δ contains the pairs $((1, a), 2)$, $((1, b), 4)$, $((2, a), 3)$, $((2, b), 7)$, $((3, a), 2)$, $((3, b), 6)$, $((4, a), 5)$, $((4, b), 3)$, $((5, a), 2)$,$((5, b), 8)$, $((6, a), 9)$, $((6, b), 5)$, $((7, a), 4)$, $((7, b), 6)$, $((8, a), 9)$, $((8, b), 3)$, $((9, a), 9)$, $((9, b), 9)$. Compute an equivalent minimal deterministic finite-state automaton.

Exercise 3.9.12 Let Σ be a finite alphabet. Prove that for every word $w \in \Sigma^*$ the minimal deterministic finite-state automaton for the language $\Sigma^* \cdot \{w\}$ has exactly $|w| + 1$ states.

Chapter 4

Monoidal multi-tape automata and finite-state transducers

An important generalization of classical finite-state automata are multi-tape automata, which are used for recognizing *relations* of a particular type. The so-called regular relations (also refered as "rational relations") have been extensively studied [Eilenberg, 1974, Sakarovitch, 2009]. These relations offer a natural way to formalize all kinds of translations and transformations, which makes multi-tape automata interesting for many practical applications and explains the general interest in this kind of device. A presentation with a focus on applications in Natural Language Processing is given in [Kaplan and Kay, 1994, Roche and Schabes, 1997b]. From the perspective developed in the previous chapters, multi-tape automata represent a special subtype of monoidal automata. A natural subclass are monoidal finite-state transducers, which can be defined as two-tape automata where the first tape reads strings. In this chapter we present the most important properties of monoidal multi-tape automata in general and monoidal finite-state transducers in particular. We have seen that many of the constructions and closure properties for classical finite-state automata like intersection, complement and determinization cannot be generalized to multi-tape automata. Still, the class of relations recognized by n-tape automata is closed under a number of useful relational operations like composition, Cartesian product, projection, inverse etc. We further present a procedure for deciding the functionality of classical finite-state transducers.

4.1 Monoidal multi-tape automata

We first introduce the general concept of a monoidal multi-tape automaton and add examples for illustration. Additional notions used to describe the behaviour have been introduced above for the full class of monoidal automata and need not to be redefined here.

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

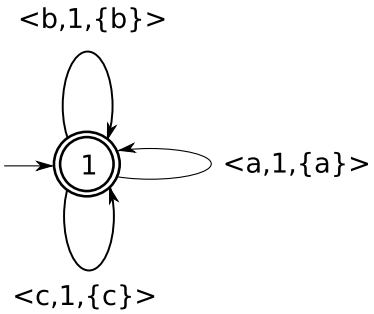

Figure 4.1: Illustration for Example 4.1.2 - a monoidal 3-tape automaton for computing the length and the set of symbols of input strings.

Definition 4.1.1 A *monoidal n-tape automaton* is a monoidal finite-state automaton $\mathcal{A} = \langle \prod_{i=1}^{n} \mathcal{M}_i, Q, I, F, \Delta \rangle$ over a monoid $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 \ldots \times \mathcal{M}_n$ which is a Cartesian product of n monoids. If $n \geq 2$, the automaton is also called a *multi-tape automaton*.

Since the class of monoids is closed under Cartesian products (cf. Definition 1.5.15) it is clear that monoidal n-tape automata are just special instances of monoidal finite-state automata in the sense of Definition 2.1.1. Hence, the definitions of an $\langle e_1, e_2, \ldots, e_n \rangle$*-free* monoidal n-tape automaton, *(successful) paths in n-tape monoidal automata*, and *monoidal n-tape automaton languages* are special instances of the more general definitions given in Section 2, assuming that the monoid $\mathcal{M}$ has the form $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 \ldots \times \mathcal{M}_n$ and unit element $\langle e_1, e_2, \ldots, e_n \rangle$.
For a monoidal n-tape automaton $\mathcal{A}$, the term "monoidal language *accepted* by $\mathcal{A}$" (cf. Definition 2.1.10) is slightly misleading in the sense that n-tape automata are not primarily used to accept or reject input. Often one tape (projection to a fixed component) is used to consume given input and to define the possible traversal paths from state to state. At each transition, the other tapes produce an output. The role of input and output tapes may be changed. In this sense, n-tape automata can be considered as devices for generating output(s) for given input, and for "translating" input sequences. Later, when looking at deterministic devices we introduce the notion of an *input* language and an "output function", which reflects this functionality (vf. Remark 5.1.3).

Example 4.1.2 Let $\Sigma := \{a, b, c\}$ denote an alphabet. We consider the three monoids $\mathcal{M}_1 := \Sigma^*$, $\mathcal{M}_2 := \langle \mathbb{N}, +, 0 \rangle$, and $\mathcal{M}_3 := \langle 2^{\Sigma}, \cup, \emptyset \rangle$. Let $\mathcal{A}$ be the monoidal automaton over $\mathcal{M}_1 \times \mathcal{M}_2 \times \mathcal{M}_3$ with just one state $q = 1$, which also represents an initial and a final state. The transition relation Δ contains the triples $\langle q, \langle a, 1, \{a\} \rangle, q \rangle$, $\langle q, \langle b, 1, \{b\} \rangle, q \rangle$, and $\langle q, \langle c, 1, \{c\} \rangle, q \rangle$. The automaton is shown in Figure 4.1. The monoidal language recognized by $\mathcal{A}$ is the relation containing all triples $\langle w, n, A \rangle$ where $w \in \Sigma^*$, $n = |w|$, and A is the set of letters occurring in w. For example, the null-path has the label $\langle \varepsilon, 0, \emptyset \rangle$, and the label of the successful path

$$1 \rightarrow^{\langle a,1,\{a\}\rangle} 1 \rightarrow^{\langle a,1,\{a\}\rangle} 1 \rightarrow^{\langle b,1,\{b\}\rangle} 1 \rightarrow^{\langle a,1,\{a\}\rangle} 1$$

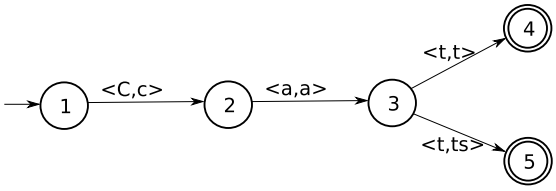

Figure 4.2: Illustration for Example 4.1.3 - a non-deterministic monoidal 2-tape automaton for generating morphological variants.

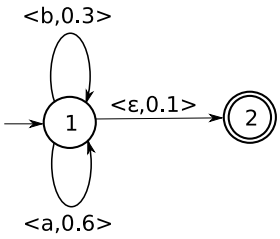

Figure 4.3: Illustration for Example 4.1.4 - a monoidal 2-tape automaton for computing the probability for generating a word.

is $\langle aaba, 4, \{a, b\}\rangle$. Following the above perspective, the input *aaba* on the first tape is translated into $4 = |aaba|$ on the second tape and into $\{a, b\}$ on the third tape. In this way $\mathcal{A}$ encodes the two homomorphisms $h_1 : w \mapsto |w|$ and $h_2 : w \mapsto \{\sigma \in \Sigma \mid \sigma \text{ occurs in } w\}$.

Example 4.1.3 Let $\Sigma := \{C, c, a, t, s\}$, let $\mathcal{A}$ be the monoidal automaton over $\Sigma^* \times \Sigma^*$ with set of states $Q = \{1, 2, 3, 4, 5\}$, set of initial states $I := \{1\}$, set of final states $F := \{4, 5\}$ and the transition relation

$$\Delta := \{\langle 1, \langle C, c\rangle, 2\rangle, \langle 2, \langle a, a\rangle, 3\rangle, \langle 3\langle t, t\rangle, 4\rangle, \langle 3, \langle t, ts\rangle, 5\rangle\}.$$

The automaton is shown in Figure 4.2. Following the above perspective, the input *Cat* on the first tape has the two possible translations *cat*, *cats*. The example demonstrates that non-deterministic two-tape automata can be used to generate orthographic and inflectional variants for a given input word.

Example 4.1.4 Sometimes words are assumed to be "generated" by a kind of stochastic process. Each letter σ of the input alphabet comes with a probability $p(\sigma)$, there is also a probability that the production of a string stops. Figure 4.1.4 shows a 2-tape automaton that computes the probability for generating words over the alphabet $\{a, b\}$. The two monoids are the free monoid over alphabet $\{a, b\}$ and the set of positive real numbers with multiplication. The probabilities for producing a letter respectively are $p(a) = 0.6$ and $p(b) = 0.3$. Path

$$1 \to^{\langle a,0.6\rangle} 1 \to^{\langle a,0.6\rangle} \to^{\langle b,0.3\rangle} \to^{\langle \varepsilon,0.1\rangle} 2$$

shows that the probability of generating *aab* under the formalized prozess is $0,6 \cdot 0.6 \cdot 0.3 \cdot 0.1$.

In order to stress that the languages accepted by monoidal multi-tape automata are relations we introduce the following terminology.

Definition 4.1.5 A *monoidal n-tape automaton relation* is a monoidal language recognized by a monoidal n-tape automaton.

4.2 Additional closure properties of monoidal multi-tape automata

In Section 2.5 we have seen that e-transitions can be removed for all monoidal finite-state automata, hence this result holds in particular for all monoidal n-tape automata. In Section 2.2 it has been shown that the class of monoidal languages recognized by monoidal finite-state automata over a fixed monoid $\mathcal{M}$ is closed under union, concatenation, and Kleene-Star. Hence these closure properties in particular hold for monoidal n-tape automata over the same monoid. We now want to show that the class of relations accepted by monoidal multi-tape automata[1] is closed under some additional operations. Some of the following constructions are obtained in a more elegant way if we add an e-loop to each state. We define

$$E(\Delta) := \Delta \cup \{\langle q, e, q\rangle \mid q \in Q\}.$$

Obviously, replacing Δ by $E(\Delta)$ will not change the language recognized by the underlying monoidal automaton.

Proposition 4.2.1

1. *(Cartesian product) Let $\mathcal{A}_1 = \langle \mathcal{M}_1, Q_1, I_1, F_1, \Delta_1\rangle$ and $\mathcal{A}_2 = \langle \mathcal{M}_2, Q_2, I_2, F_2, \Delta_2\rangle$ be two monoidal automata and let*

   $$\Delta := \{\langle \langle q_1, q_2\rangle, \langle u_1, u_2\rangle, \langle q_1', q_2'\rangle\rangle \mid \langle q_1, u_1, q_1'\rangle \in E(\Delta_1) \;\&\; \langle q_2, u_2, q_2'\rangle \in E(\Delta_2)\}.$$

   Then for the monoidal 2-tape automaton

   $$\mathcal{A} := \langle \mathcal{M}_1 \times \mathcal{M}_2, Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \Delta\rangle$$

   we have $L(\mathcal{A}) = L(\mathcal{A}_1) \times L(\mathcal{A}_2)$.

2. *(Projection) Let $\mathcal{A} = \langle \mathcal{M}_1 \times \mathcal{M}_2 \times \ldots \times \mathcal{M}_n, Q, I, F, \Delta\rangle$ be a monoidal n-tape automaton and $n \geq 2$. Let $\Delta_{\times i} := \{\langle q, \bar{u}_{\times i}, q'\rangle \mid \langle q, \bar{u}, q'\rangle \in \Delta\}$. Then for the monoidal $(n-1)$-tape automaton*

   $$\mathcal{A}' := \langle \mathcal{M}_1 \times \ldots \times \mathcal{M}_{i-1} \times \mathcal{M}_{i+1} \times \ldots \times \mathcal{M}_n, Q, F, I, \Delta_{\times i}\rangle$$

   we have $L(\mathcal{A}') = L(\mathcal{A})_{\times i}$.

3. *(Inverse relation for 2-tape automata) Let $\mathcal{A} = \langle \mathcal{M}_1 \times \mathcal{M}_2, Q, I, F, \Delta\rangle$ be a monoidal 2-tape automaton and*

   $$\Delta' := \{\langle q_1, \langle v, u\rangle, q_2\rangle \mid \langle q_1, \langle u, v\rangle, q_2\rangle \in \Delta\}.$$

[1] We do not always fix the number n.

Then for the monoidal 2*-tape automaton*

$$\mathcal{A}' = \langle \mathcal{M}_2 \times \mathcal{M}_1, Q, I, F, \Delta' \rangle$$

we have $L(\mathcal{A}') = L(\mathcal{A})^{-1}$.

4. *(Identity relation) Let* $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ *be a monoidal automaton and*

$$\Delta' := \{\langle q_1, \langle u, u \rangle, q_2 \rangle \mid \langle q_1, u, q_2 \rangle \in \Delta\}.$$

Then for the monoidal 2*-tape automaton*

$$\mathcal{A}' = \langle \mathcal{M} \times \mathcal{M}, Q, I, F, \Delta' \rangle$$

we have $L(\mathcal{A}') = Id_{L(\mathcal{A})}$.

Proof. (Sketch) The simple proofs for Parts 2-4 are omitted. As to Part 1, let $m_1 \in L(\mathcal{A}_1)$ and $m_2 \in L(\mathcal{A}_2)$. Then there exist two successful paths in $\mathcal{A}_1$ and $\mathcal{A}_2$ of the form

$$\begin{array}{llll} q_1^1 \to & \dots & \to q_r^1 \\ q_1^2 \to & \dots & \to q_s^2 \end{array}$$

with labels m_1 and m_2, respectively. If the two paths have distinct length, then we may add transitions of the form $\langle q, e, q \rangle$ (from one of the sets $E(\Delta_1)$, $E(\Delta_2)$) to the shorter path such that both paths have the same length. Note that the label of the extended path is not modified. Once the two paths have the same length, we may combine the first (second,..) transitions of the two paths to a transition of Δ. We obtain a successful path for $\mathcal{A}$ with label $\langle m_1, m_2 \rangle$. This shows that $L(\mathcal{A}_1) \times L(\mathcal{A}_2) \subseteq L(\mathcal{A})$. Conversely, from a successful path in $\mathcal{A}$ with label $\langle m_1, m_2 \rangle$, using obvious projections we can reconstruct two successful paths with labels m_1 and m_2 where transitions respectively are in $E(\Delta_1)$ and $E(\Delta_2)$. Leaving out the transitions of the form $\langle q, e, q \rangle$ we obtain successful paths in $\mathcal{A}_1$ and $\mathcal{A}_2$ with labels m_1 and m_2. □

Corollary 4.2.2 *The class of monoidal multi-tape automaton relations is closed under Cartesian products, projections, and inverse relations.*

Remark 4.2.3 In Remark 1.5.16 we have seen that projection is a monoid homomorphism. The inversion of relations is a monoid homomorphism $\mathcal{M}_1 \times \mathcal{M}_2 \to \mathcal{M}_2 \times \mathcal{M}_1$. Also the identity function is a monoid homomorphism $\mathcal{M}_1 \to \mathcal{M}_1 \times \mathcal{M}_1$. Hence the second, the third and the fourth construction in Proposition 4.2.1 are special cases of our earlier observation that monoidal languages accepted by monoidal finite-state automata are closed under homomorphic images.

4.3 Classical multi-tape automata and letter automata

Definition 4.3.1 A *classical* n-tape automaton is an n-tape automaton over a monoid that is a Cartesian product of free monoids.

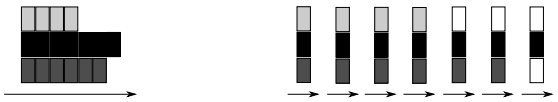

Figure 4.4: Illustration for the proof of Proposition 4.3.3. A single transition of an automaton with three tapes shown on the left-hand side is split into several transitions with simple labels (letters or empty word) on the right-hand side. Grey tones correspond to tapes, grey boxes to letters, white boxes to ε.

Classical n-tape automata are often written in the simpler form

$$\langle \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n, Q, I, F, \Delta \rangle ,$$

specifying the underlying alphabets of the free component monoids. Note that in a classical n-tape automaton the components of the transition labels are arbitrary strings over the given alphabet. A formally more restricted kind of classical n-tape automaton is captured by the following definition.

Definition 4.3.2 An *n-tape letter automaton* is a classical n-tape automaton $\mathcal{A} = \langle \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n, Q, I, F, \Delta \rangle$ such that $\Delta \subseteq Q \times (\Sigma_1^\varepsilon \times \ldots \times \Sigma_n^\varepsilon) \times Q$.

In a letter automaton each component of a transition label is either a single symbol or the empty word. As to the recognition power, there is no difference between classical n-tape automata and n-tape letter automata.

Proposition 4.3.3 *Let $\mathcal{A}$ be a classical n-tape automaton. Then $\mathcal{A}$ can be converted to an equivalent n-tape letter automaton.*

Proof. We substitute each transition

$$\langle q', \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle , q'' \rangle \in \Delta$$

and $k = \max_{i=1,\ldots,n} |\alpha_i|$ where $k > 1$ with a sequence of k transitions of the form

$$\left\langle q', \left\langle \alpha_1^1, \ldots, \alpha_n^1 \right\rangle , q_1' \right\rangle , \left\langle q_1', \left\langle \alpha_1^2, \ldots, \alpha_n^2 \right\rangle , q_2' \right\rangle , \ldots , \left\langle q_{k-1}', \left\langle \alpha_1^k, \ldots, \alpha_n^k \right\rangle , q'' \right\rangle$$

where $q_1', q_2', \ldots, q_{k-1}'$ are new non-final states and α_j^i is the i-th symbol of α_j if $i \leq |\alpha_j|$ and $\alpha_j^i = \varepsilon$ otherwise. □

The construction principle used in the above proof is illustrated in Figure 4.4.

Example 4.3.4 The automaton shown in Figure 2.4 is a two-tape letter automaton. The automaton in Figure 4.2 is a classical two-tape automaton. It is not a letter automaton since one transition has a label $\langle t, ts \rangle$.

Further closure properties for classical n-tape automata

Classical n-tape automata have additional closure properties. For the following constructions it is important that the components of transition tuples are always single symbols or the empty word. Hence we demand that the input automata are letter automata.

Proposition 4.3.5 *(Relational composition) Let*

$$\begin{aligned}\mathcal{A}_1 &= \langle \Sigma_1 \times \Sigma, Q_1, I_1, F_1, \Delta_1 \rangle \\ \mathcal{A}_2 &= \langle \Sigma \times \Sigma_2, Q_2, I_2, F_2, \Delta_2 \rangle\end{aligned}$$

be 2-tape letter automata. Let Δ be the set of all tuples

$$\langle \langle q_1, q_2 \rangle, \langle u, w \rangle, \langle q_1', q_2' \rangle \rangle$$

where there exists $v \in \Sigma \cup \{\varepsilon\}$ such that $\langle q_1, \langle u, v \rangle, q_1' \rangle \in E(\Delta_1)$ and $\langle q_2, \langle v, w \rangle, q_2' \rangle \in E(\Delta_2)$. Then for the 2-tape automaton

$$\mathcal{A} := \langle \Sigma_1 \times \Sigma_2, Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \Delta \rangle$$

we have $L(\mathcal{A}) = L(\mathcal{A}_1) \circ L(\mathcal{A}_2)$ (here $\circ$ denotes relational composition, cf. Def. 1.1.3).

Note that relational composition only makes sense if the alphabet of tape 2 of $\mathcal{A}_1$ coincides with the alphabet of tape 1 of $\mathcal{A}_2$ as above.

Proof. We have to prove that $L(\mathcal{A}) = L(\mathcal{A}_1) \circ L(\mathcal{A}_2)$.

("$\subseteq$") Let $\langle \alpha, \gamma \rangle \in L(\mathcal{A})$. Then there exists a successful path in $\mathcal{A}$ of the form

$$\pi = \langle q_0', q_0'' \rangle \to^{\langle \alpha_1, \gamma_1 \rangle} \langle q_1', q_1'' \rangle \ldots \to^{\langle \alpha_k, \gamma_k \rangle} \langle q_k', q_k'' \rangle ,$$

such that $\langle \alpha_1, \gamma_1 \rangle \bar{\cdot} \ldots \bar{\cdot} \langle \alpha_k, \gamma_k \rangle = \langle \alpha, \gamma \rangle$. The definition of the transition relation Δ of $\mathcal{A}$ implies that there exists a sequence $\beta_1, \beta_2, \ldots, \beta_k \in \Sigma^\varepsilon$ such that $\langle q_{i-1}', \langle \alpha_i, \beta_i \rangle, q_i' \rangle \in E(\Delta_1)$ and $\langle q_{i-1}'', \langle \beta_i, \gamma_i \rangle, q_i'' \rangle \in E(\Delta_1)$ for $i = 1, \ldots, k$. The two paths

$$\begin{aligned}\pi' &= q_0' \to^{\langle \alpha_1, \beta_1 \rangle} q_1' \ldots \to^{\langle \alpha_k, \beta_k \rangle} q_k' \\ \pi'' &= q_0'' \to^{\langle \beta_1, \gamma_1 \rangle} q_1' \ldots \to^{\langle \beta_k, \gamma_k \rangle} q_k''\end{aligned}$$

are successful paths in $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. Hence for $\beta = \beta_1 \cdot \ldots \cdot \beta_k$ we have $\langle \alpha, \beta \rangle \in L(\mathcal{A}_1)$ and $\langle \beta, \gamma \rangle \in L(\mathcal{A}_2)$.

("$\supseteq$") Let $\langle \alpha, \beta \rangle \in L(\mathcal{A}_1)$, $\langle \beta, \gamma \rangle \in L(\mathcal{A}_2)$, let $l := |\beta|$. Then there exists a pair of successful paths

$$\begin{aligned}\pi' &= q_0' \to^{\langle \alpha_1, \beta_1' \rangle} q_1' \ldots \to^{\langle \alpha_k, \beta_k' \rangle} q_k' \\ \pi'' &= q_0'' \to^{\langle \beta_1'', \gamma_1 \rangle} q_1'' \ldots \to^{\langle \beta_m'', \gamma_m \rangle} q_m''\end{aligned}$$

such that $\langle \alpha_1, \beta_1' \rangle \cdot \ldots \cdot \langle \alpha_k, \beta_k' \rangle = \langle \alpha, \beta \rangle$, $\langle \beta_1'', \gamma_1 \rangle \cdot \ldots \cdot \langle \beta_m'', \gamma_m \rangle = \langle \beta, \gamma \rangle$, $q_0' \in I_1, q_0'' \in I_2, q_k' \in F_1, q_m'' \in F_2$ and $k, m \geq l$. Recall that all transition labels β_i' and β_j'' on the common tape (second tape of $\mathcal{A}_1$, first tape of $\mathcal{A}_2$) are single letters or the empty string ε. The two paths can have distinct lengths, and at distinct positions we may have transitions with label ε on the common tape. We now show that using identity transitions $\langle q, \langle \varepsilon, \varepsilon \rangle, q \rangle$ from $E(\Delta_1) \cup E(\Delta_2)$ it is possible to "synchronize" both paths in order to define a suitable path of $\mathcal{A}$.

Since $\beta_1' \cdot \ldots \cdot \beta_k' = \beta_1'' \cdot \ldots \cdot \beta_m''$, there exist two subsets of indices $\{i_1, i_2, \ldots, i_l\} \subseteq \{1, \ldots, k\}$ and $\{j_1, j_2, \ldots, j_l\} \subseteq \{1, \ldots, m\}$, such that $\beta_{i_1}' = \beta_{j_1}'' \in \Sigma, \ldots, \beta_{i_l}' = \beta_{j_l}'' \in \Sigma$ and $\forall i \in \{1, \ldots, k\} \setminus \{i_1, i_2, \ldots, i_l\} : \beta_i' = \varepsilon$ and $\forall j \in \{1, \ldots, m\} \setminus$

$\{j_1, j_2, \ldots, j_l\} : \beta''_j = \varepsilon$. Then we construct the following synchronized path π in $\mathcal{A}$ - arrows "$\longrightarrow$" indicate that at this step we consume a letter on each tape:

$$\begin{aligned}
\pi = \langle q'_0, q''_0\rangle &\to^{\langle \alpha_1, \varepsilon\rangle} \langle q'_1, q''_0\rangle \ldots \to^{\langle \alpha_{i_1-1}, \varepsilon\rangle} \langle q'_{i_1-1}, q''_0\rangle \\
&\to^{\langle \varepsilon, \gamma_1\rangle} \langle q'_{i_1-1}, q''_1\rangle \ldots \to^{\langle \varepsilon, \gamma_{j_1-1}\rangle} \langle q'_{i_1-1}, q''_{j_1-1}\rangle \\
&\longrightarrow^{\langle \alpha_{i_1}, \gamma_{j_1}\rangle} \langle q'_{i_1}, q''_{j_1}\rangle \\
&\to^{\langle \alpha_{i_1+1}, \varepsilon\rangle} \langle q'_{i_1+1}, q''_{j_1}\rangle \ldots \to^{\langle \alpha_{i_2-1}, \varepsilon\rangle} \langle q'_{i_2-1}, q''_{j_1}\rangle \\
&\to^{\langle \varepsilon, \gamma_{i_1+1}\rangle} \langle q'_{i_2-1}, q''_{i_1+1}\rangle \ldots \to^{\langle \varepsilon, \gamma_{j_2-1}\rangle} \langle q'_{i_2-1}, q''_{j_2-1}\rangle \\
&\longrightarrow^{\langle \alpha_{i_2}, \gamma_{j_2}\rangle} \langle q'_{i_2}, q''_{j_2}\rangle \\
&\vdots \\
&\to^{\langle \alpha_{i_{l-1}+1}, \varepsilon\rangle} \left\langle q'_{i_{l-1}+1}, q''_{j_{l-1}}\right\rangle \ldots \to^{\langle \alpha_{i_l-1}, \varepsilon\rangle} \left\langle q'_{i_l-1}, q''_{j_{l-1}}\right\rangle \\
&\to^{\langle \varepsilon, \gamma_{i_{l-1}+1}\rangle} \left\langle q'_{i_l-1}, q''_{i_{l-1}+1}\right\rangle \ldots \to^{\langle \varepsilon, \gamma_{j_l-1}\rangle} \langle q'_{i_l-1}, q''_{j_l-1}\rangle \\
&\longrightarrow^{\langle \alpha_{i_l}, \gamma_{j_l}\rangle} \langle q'_{i_l}, q''_{j_l}\rangle \\
&\to^{\langle \alpha_{i_l+1}, \varepsilon\rangle} \langle q'_{i_l+1}, q''_{j_l}\rangle \ldots \to^{\langle \alpha_k, \varepsilon\rangle} \langle q'_k, q''_{j_l}\rangle \\
&\to^{\langle \varepsilon, \gamma_{i_l+1}\rangle} \langle q'_k, q''_{i_l+1}\rangle \ldots \to^{\langle \varepsilon, \gamma_m\rangle} \langle q'_k, q''_m\rangle
\end{aligned}$$

Clearly π is a successful path of $\mathcal{A}$ and its label is $\langle \alpha, \gamma\rangle$. □

Proposition 4.3.6 *(n-way reverse relation) Let*

$$\mathcal{A} = \langle \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n, Q, I, F, \Delta\rangle$$

be a classical n-tape automaton. Let $\Delta' = \{\langle q_2, \bar{\rho}(\bar{a}), q_1\rangle \mid \langle q_1, \bar{a}, q_2\rangle \in \Delta\}$. *Then for the n-tape automaton*

$$\mathcal{A} = \langle \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^*, Q, F, I, \Delta'\rangle$$

we have $R(\mathcal{A}) = \bar{\rho}(R(\mathcal{A}_1))$.

Proof. Similarly as in Proposition 3.3.4 the proof of the property follows directly from the definitions.

4.4 Monoidal finite-state transducers

In the literature there are distinct notions of finite-state transducers. Typically these notions refer to 2-tape finite-state devices that are used for translational tasks: an input string, to be consumed on the first tape, is translated into an output element given by the composition of the corresponding labels of the second tape. Some authors use the notion only for deterministic devices (we do not follow this line). We now generalize this concept. Monoidal transducers in our

sense also have two tapes. Since monoidal transducers are meant to represent translation devices we always demand that the first tape reads conventional strings over a finite alphabet as opposed to elements of an arbitrary monoid. However, the second tape may run over any monoid. In this section, after introducing monoidal finite-state transducers we define some notions that become relevant when we want to determinize transducers or simplify the structure.

Definition 4.4.1 A *monoidal finite-state transducer* is a monoidal 2-tape automaton $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ where the underlying product monoid has the form $\Sigma^* \times \mathcal{M}$ for some alphabet Σ. If $\Delta \subseteq Q \times (\Sigma^\varepsilon \times \mathcal{M}) \times Q$, then $\mathcal{T}$ is called a monoidal *letter transducer*. A variant of Proposition 4.3.3 is the following.

Proposition 4.4.2 *Each monoidal finite-state transducer can be converted to an equivalent monoidal letter transducer.*

Proof. This is an obvious variant of the proof of Proposition 4.3.3. □

For example, the 2-tape automata shown in Figures 4.3 and 4.2 are monoidal letter transducers. Note that finite-state transducers in our sense in general are *non-deterministic* machines, which means that an input string can be translated to distinct output elements. We shall look at deterministic variants of the concept in the following chapter. In a monoidal finite-state transducer, the first component of each transition label is a letter or the empty string. Hence, when ignoring the second components we obtain a classical finite-state automaton.

Definition 4.4.3 Let $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal finite-state transducer. Let

$$\Delta_1 = \{\langle p, w, q \rangle \mid \exists m \in M : \langle p, \langle w, m \rangle, q \rangle \in \Delta\}.$$

Then $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta_1 \rangle$ is called the *underlying finite-state automaton* of $\mathcal{T}$.

The underlying automaton is the projection of $\mathcal{T}$ (cf. Part 2 of Proposition 4.2.1) where we "cross out" the second tape. The following definitions become relevant when trying to determinize transducers.

Definition 4.4.4 A monoidal finite-state transducer $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ is *functional* iff the language $L(\mathcal{T})$ represents a partial function $\Sigma^* \to \mathcal{M}$. In this situation, $\mathcal{T}$ is said to *represent* the function $L(\mathcal{T})$.

Definition 4.4.5 A monoidal finite-state transducer $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ is *infinitely ambiguous* if the relation $L(\mathcal{T})$ is infinitely ambiguous (cf. Definition 1.1.6).

Clearly, functional transducers are not infinitely ambiguous. Following Definition 2.1.12, monoidal multi-tape automata are called equivalent if they have the same monoidal language. In what follows we want to simplify a given monoidal finite-state transducer, say, with output in the monoid $\langle M, \circ, e \rangle$, in the sense that in the new transducer at each transition step the first tape always reads a letter. In other words, we want to get rid of transitions where the first component of the transition label is the empty word.

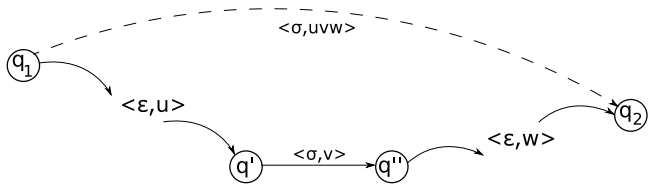

Figure 4.5: Illustration for ε-removal in transducers. New transition links (dashed line) with joined output are added to Δ' before ε-transitions are removed.

Definition 4.4.6 A monoidal finite-state transducer $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ is said to be *real-time* if $\Delta \subseteq Q \times (\Sigma \times \mathcal{M}) \times Q$.

Obviously, a real-time transducer cannot produce output distinct from e for the empty input string. Hence, if the language of a given source transducer contains a pair $\langle \varepsilon, m \rangle$ such that $m \neq e$, we cannot find an equivalent real-time transducer. This motivates the following definition.

Definition 4.4.7 Two monoidal finite-state transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ over the same monoid monoid $\langle M, \circ, e \rangle$ are *equivalent up to* ε if $L(\mathcal{T}_1) \setminus (\{\varepsilon\} \times M) = L(\mathcal{T}_2) \setminus (\{\varepsilon\} \times M)$.

Proposition 4.4.8 *Let $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal letter transducer. Assume that the set of all path labels of the form $\langle \varepsilon, m \rangle$ in $\mathcal{T}$ is finite. Then there exists a real-time transducer $\mathcal{T}'$ equivalent to $\mathcal{T}$ up to ε.*

Proof. Similarly as in the construction presented in Proposition 2.5.6 we introduce a new set of transitions Δ' which consists of all transitions of the form $\langle q_1, \langle \sigma, uvw \rangle, q_2 \rangle$ such that $\sigma \in \Sigma$, there exist states $q', q'' \in Q$ and entries $\langle q_1, \langle \varepsilon, u \rangle, q' \rangle \in \Delta^*$, $\langle q', \langle \sigma, v \rangle, q'' \rangle \in \Delta$ and $\langle q'', \langle \varepsilon, w \rangle, q_2 \rangle \in \Delta^*$. See the illustration in Figure 4.5 for the construction of Δ'. Our assumption on path labels ensures that Δ' is finite. Consider the transducer $\mathcal{T}' = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta' \rangle$. For each successful path of $\mathcal{T}$ with non-empty input we find a corresponding successful path of $\mathcal{T}'$ with the same input and output. The same holds in the converse direction. It follows that $\mathcal{T}'$ is equivalent to $\mathcal{T}$ up to ε. □

As a matter of fact the above condition on path labels always holds if the monoidal letter transducer $\mathcal{T}$ does not have any ε-cycle.

4.5 Classical finite-state transducers

Definition 4.5.1 A *classical* finite-state transducer is a transducer

$$\mathcal{T} = \langle \Sigma_1^* \times \Sigma_2^*, Q, I, F, \Delta \rangle$$

where the second tape runs over a free monoid Σ_2^*.

Classical *letter* transducers are defined in accordance with Definition 4.4.1. When looking at classical transducers, the following notions are useful.

Definition 4.5.2 A *binary regular string relation* is a binary string relation in the sense of Definition 1.4.6 which is regular (i.e., a monoidal regular 2-relation in the sense of Definition 2.3.4).

Example 4.5.3 Let $\Sigma = \{\sigma_1 \ldots, \sigma_n\}$ be an alphabet, let $L \subseteq \Sigma^*$ be a regular language. Then

1. $Id(\Sigma^*)$ is a binary regular string relation. We have $Id(\Sigma^*) = (\{\langle \sigma_1, \sigma_1 \rangle\} \cup \ldots \cup \{\langle \sigma_n, \sigma_n \rangle\})^*$.
2. $Id(L)$ is a binary regular string relation.

As to Point 1 note that $Id(\Sigma^*) = (\{\langle \sigma_1, \sigma_1 \rangle\} \cup \ldots \cup \{\langle \sigma_n, \sigma_n \rangle\})^*$. The proof of Point 2 is left as an exercise (cf. Exercise 4.8.5).

The following examples show that binary regular string relations can be infinitely ambiguous.

Example 4.5.4 Let $\Sigma = \{a, b\}$. Then $R_1 := \{\langle \varepsilon, b^n \rangle \mid n \in \mathbb{N}\}$ is a binary regular string relation. We have $R_1 = \{\langle \varepsilon, b \rangle\}^*$. Also $R_2 := \{\langle a, b^n \rangle \mid n \in \mathbb{N}\} = \{\langle a, \varepsilon \rangle\} \cdot R_1$ is an infinitely ambigious binary string relation.

The following characterization is a direct consequence of Theorem 2.4.2.

Theorem 4.5.5 *The binary regular string relations are exactly the relations accepted by classical finite-state transducers.*

Definition 4.5.6 A *regular string function* is a binary regular string relation that is a partial function.

In the literature, regular string functions are also called *rational functions.* In the following chapters we study how regular string functions can be recognized by means of deterministic finite-state devices. At this point, the following observations become relevant. Recall Definition 4.4.5.

Proposition 4.5.7 *A trimmed classical finite-state transducer $\mathcal{T}$ is infinitely ambiguous iff there exists a loop in $\mathcal{T}$ with a label $\langle \varepsilon, u \rangle$, where $u \neq \varepsilon$.*

Proof. Let $\mathcal{T} = \langle \Sigma_1^* \times \Sigma_2^*, Q, I, F, \Delta \rangle$. If a loop of the above form exists, then the transducer is obviously infinitely ambiguous.
Let us now assume that the transducer is infinitely ambiguous. Without loss of generality we assume that the transducer is $\langle \varepsilon, \varepsilon \rangle$-free. Let $\alpha \in \Sigma_1^*$ be a string such that the set of translations $L(\mathcal{T}) \cap (\{\alpha\} \times \Sigma_2^*)$ is infinite. Then there exists an infinite number of successful paths with a label of the form $\langle \alpha, \beta \rangle$. Since the set of transitions is finite there exists a successful path π of this form of length $> (|\alpha| + 1)|Q|$. Since at most $|\alpha|$ transitions on the path have a nonempty word as input, there exists a subpath π' in π of length $> |Q|$ that has the empty word as input label. Therefore there exists a subpath π'' of π' which represents a loop. The input label is ε and since there are no $\langle \varepsilon, \varepsilon \rangle$ transitions its output label is non-empty. □

Remark 4.5.8 If $\mathcal{T}$ is a trimmed classical finite-state transducer and $\mathcal{T}$ is not infinitely ambiguous, then it follows from Proposition 4.5.7 that the set of all path labels of the form $\langle \varepsilon, w \rangle$ in $\mathcal{T}$ is finite. Hence we can build a real-time transducer $\mathcal{T}'$ equivalent to $\mathcal{T}$ up to ε.

4.6 Deciding functionality of classical finite-state transducers

A necessary precondition needed for the determinization of transducers (to be described in Chapter 5 and Chapter 6) is the functionality of the transducer language. Schützenberger [Schützenberger, 1975] was the first to show that it is decidable if a given transducer is functional. In this section we present a procedure for deciding the functionality of a given classical finite-state transducer based on the approach presented in [Béal et al., 2003]. We first show how to decide the functionality of classical real-time finite-state transducers. At the end of the section we consider the general case of classical finite-state transducers.

Functionality means that if an input string can be processed on distinct successful paths, the outputs of all paths must be identical. Partial initial outputs must be compatible in the sense that they have a joint extension. We start looking at a function related to the notion of the longest common prefix of two strings and its properties. This function is later used to check compatibility of initial outputs and identity of full outputs for the same input string.

Definition 4.6.1 Let Σ be a finite alphabet. The *advance function* $\omega : (\Sigma^* \times \Sigma^*) \times (\Sigma^* \times \Sigma^*) \to \Sigma^* \times \Sigma^*$ is defined as:

$$\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle) = \left\langle c^{-1}x\alpha, c^{-1}y\beta \right\rangle, \text{ where } c = x\alpha \wedge y\beta.$$

The *iterated advance function* $\omega^* : (\Sigma^* \times \Sigma^*) \times (\Sigma^* \times \Sigma^*)^* \to (\Sigma^* \times \Sigma^*)$ is defined inductively (note that ε and U denote sequences of pairs of strings):

- $\omega^*(\langle x, y\rangle, \varepsilon) = \langle x, y\rangle$,
- $\omega^*(\langle x, y\rangle, U\,\langle \alpha, \beta\rangle) = \omega(\omega^*(\langle x, y\rangle, U), \langle \alpha, \beta\rangle)$.

We say that $\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle)$ (resp. $\omega^*(\langle x, y\rangle, U)$) is the *advance* of $\langle x, y\rangle$ with $\langle \alpha, \beta\rangle$ (resp. U).
The advance $\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle) = \langle u, v\rangle$ of the pair of words $\langle x, y\rangle \in \Sigma^* \times \Sigma^*$ with $\langle \alpha, \beta\rangle \in \Sigma^* \times \Sigma^*$ is said to be *balancible* if $u = \varepsilon$ or $v = \varepsilon$.

Example 4.6.2 Since $abbc = abbc \wedge abbccc$ we have

$$\omega(\langle abb, \varepsilon\rangle, \langle c, abbccc\rangle) = \langle (abbc)^{-1}abbc, (abbc)^{-1}abbccc\rangle = \langle \varepsilon, cc\rangle$$

and since $ccdd \wedge ccd = ccd$ we have

$$\omega^*(\langle abb, \varepsilon\rangle, \langle c, abbccc\rangle\,\langle ccdd, d\rangle) = \omega^*(\langle \varepsilon, cc\rangle, \langle ccdd, d\rangle) = \langle d, \varepsilon\rangle.$$

Intuitively, the (iterated) advance function is meant to measure for two strings "growing to the right" how much the two strings differ when ignoring the longest common prefix. If the advance is balancible, then the advance function shows how much one of the strings is "ahead" of the other string. If the strings are not balancible, then they cannot be extended to the same string. The next proposition formalizes these properties.

Proposition 4.6.3 *For all $x, y, \alpha', \beta', \alpha'', \beta'' \in \Sigma^*$:*

1. $\omega(\omega(\langle x, y\rangle, \langle \alpha', \beta'\rangle), \langle \alpha'', \beta''\rangle) = \omega(\langle x, y\rangle, \langle \alpha'\alpha'', \beta'\beta''\rangle)$.
2. $\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle) = \langle \varepsilon, \varepsilon\rangle$ *iff* $x\alpha = y\beta$.
3. *If the advance $\langle u, v\rangle$ of $\langle x, y\rangle$ with $\langle \alpha, \beta\rangle$ is balancible (i.e. $\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle) = \langle u, v\rangle$ and $u = \varepsilon$ or $v = \varepsilon$), then $x \cdot \alpha \cdot v = y \cdot \beta \cdot u$.*
4. *If the advance $\langle u, v\rangle$ of $\langle x, y\rangle$ with $\langle \alpha, \beta\rangle$ is not balancible (i.e. $\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle) = \langle u, v\rangle$ and $u \neq \varepsilon$ and $v \neq \varepsilon$), then for all $z, w \in \Sigma^*$ we have $x \cdot \alpha \cdot z \neq y \cdot \beta \cdot w$.*

Proof. (1.) Let $c := x\alpha' \wedge y\beta'$. Using the definition of the advance function and Proposition 1.3.6, Properties 1, 2 we obtain

$$\begin{aligned}
& \omega(\langle x, y\rangle, \langle \alpha'\alpha'', \beta'\beta''\rangle) \\
= \; & \left\langle (x\alpha'\alpha'' \wedge y\beta'\beta'')^{-1} x\alpha'\alpha'', (x\alpha'\alpha'' \wedge y\beta'\beta'')^{-1} y\beta'\beta'' \right\rangle \\
= \; & \left\langle (cc^{-1}x\alpha'\alpha'' \wedge cc^{-1}y\beta'\beta'')^{-1} cc^{-1}x\alpha'\alpha'', (cc^{-1}x\alpha'\alpha'' \wedge cc^{-1}y\beta'\beta'')^{-1} cc^{-1}y\beta'\beta'' \right\rangle \\
=^{1} \; & \left\langle (c(c^{-1}x\alpha'\alpha'' \wedge c^{-1}y\beta'\beta''))^{-1} cc^{-1}x\alpha'\alpha'', (c(c^{-1}x\alpha'\alpha'' \wedge c^{-1}y\beta'\beta''))^{-1} cc^{-1}y\beta'\beta'' \right\rangle \\
=^{2} \; & \left\langle (c^{-1}x\alpha'\alpha'' \wedge c^{-1}y\beta'\beta'')^{-1} c^{-1}x\alpha'\alpha'', (c^{-1}x\alpha'\alpha'' \wedge c^{-1}y\beta'\beta'')^{-1} c^{-1}y\beta'\beta'' \right\rangle \\
= \; & \omega(\left\langle c^{-1}x\alpha', c^{-1}y\beta' \right\rangle, \langle \alpha'', \beta''\rangle) \\
= \; & \omega(\omega(\langle x, y\rangle, \langle \alpha', \beta'\rangle), \langle \alpha'', \beta''\rangle).
\end{aligned}$$

(2.) Clearly, if $x\alpha = y\beta$, then $\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle) = \langle \varepsilon, \varepsilon\rangle$. If $x\alpha \neq y\beta$ and $c = x\alpha \wedge y\beta$, then either $c \neq x\alpha$ or $c \neq y\beta$ and therefore either $u = c^{-1}x\alpha \neq \varepsilon$ or $v = c^{-1}y\beta \neq \varepsilon$.
(3.) Without loss of generality we assume that $u = \varepsilon$. Then $x\alpha \wedge y\beta = x\alpha$ and $v = (x\alpha)^{-1}y\beta$. Therefore $x\alpha v = x\alpha(x\alpha)^{-1}y\beta = y\beta = y\beta u$.
(4.) Let $c := x\alpha \wedge y\beta$ and $c^{-1}x\alpha = u \neq \varepsilon$, $c^{-1}y\beta = v \neq \varepsilon$ and $z, w \in \Sigma^*$. Proposition 1.3.6, Properties 3, 4 show that $uz \wedge vw = \varepsilon$ and therefore $\omega(\langle u, v\rangle, \langle z, w\rangle) = \langle uz, vw\rangle \neq \langle \varepsilon, \varepsilon\rangle$. From (1.) we obtain $\omega(\langle u, v\rangle, \langle z, w\rangle) = \omega(\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle), \langle z, w\rangle) = \omega(\langle x, y\rangle, \langle \alpha z, \beta w\rangle) \neq \langle \varepsilon, \varepsilon\rangle$. Hence from (2.) we obtain $x\alpha z \neq y\beta w$.
□

After this preparation we now look at the problem to decide if a given classical real-time finite-state transducer is functional.

Definition 4.6.4 Let $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta\rangle$ be a classical real-time finite-state transducer ($\Delta \subseteq Q \times (\Sigma_I \times \Sigma^*) \times Q$). The *squared output transducer* of $\mathcal{T}$ is the classical 2-tape automaton

$$\mathcal{S}_{\mathcal{T}} = \langle \Sigma^* \times \Sigma^*, Q \times Q, I \times I, F \times F, \Delta'\rangle$$

where Δ' is the set of all transitions of the form

$$\langle \langle q_1', q_2'\rangle, \langle \alpha_1, \alpha_2\rangle, \langle q_1'', q_2''\rangle\rangle$$

where $a \in \Sigma$, $\langle q_1', \langle a, \alpha_1\rangle, q_1''\rangle \in \Delta$, and $\langle q_2', \langle a, \alpha_2\rangle, q_2''\rangle \in \Delta$.

The squared output transducer for a transducer $\mathcal{T}$ combines pairs of paths in $\mathcal{T}$ given by the *same input sequence*. However, the common input is suppressed and only the two output strings are represented. See Figure 4.6 for an illustration.

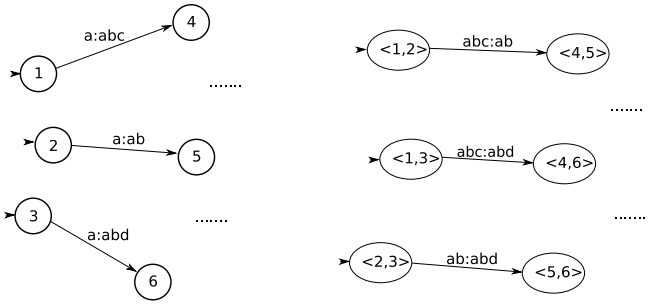

Figure 4.6: Illustration of the squared output transducer for a classical transducer. The classical transducer partially shown on the left-hand side has three transitions from initial states 1, 2, 3.

Proposition 4.6.5 *Let $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta \rangle$ be a classical real-time finite-state transducer $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta \rangle$. Then $\mathcal{T}$ is functional iff for each successful path with label $\langle \alpha, \beta \rangle$ in the squared output transducer $\mathcal{S}_\mathcal{T}$ we have $\alpha = \beta$.*

Proof. Let $\mathcal{T}$ be as in the proposition. $\mathcal{T}$ is functional iff for all $u \in \Sigma^*$ and all pairs of the form $\langle u, \alpha \rangle, \langle u, \beta \rangle \in L(\mathcal{T})$ we have $\alpha = \beta$. For $u = \varepsilon$ the property is true since $\mathcal{T}$ is real-time and therefore $\alpha = \beta = \varepsilon$. Let $|u| > 0$. If $\langle u, \alpha \rangle, \langle u, \beta \rangle \in L(\mathcal{T})$, then there exist two successful paths in $\mathcal{T}$ with labels $\langle u, \alpha \rangle$ and $\langle u, \beta \rangle$. Since $\mathcal{T}$ is real-time the label of the i-th transition on each of the two paths is the i-th letter of u and the two outputs of the paths give rise to a successful path with label $\langle \alpha, \beta \rangle$ in $\mathcal{S}_\mathcal{T}$. Conversely, if there exists a successful path with label $\langle \alpha, \beta \rangle$ in $\mathcal{S}_\mathcal{T}$, then for some $u \in \Sigma^*$ we have $\langle u, \alpha \rangle, \langle u, \beta \rangle \in L(\mathcal{T})$. Hence $\mathcal{T}$ is functional iff for each successful path in $\mathcal{S}_\mathcal{T}$ with label $\langle \alpha, \beta \rangle$ we have $\alpha = \beta$. □

Definition 4.6.6 Let $\mathcal{T} = \langle \Sigma^* \times \Sigma^*, Q, I, F, \Delta \rangle$ be a classical finite-state transducer. The pair $\langle u, v \rangle \in \Sigma^* \times \Sigma^*$ is called an *admissible advance* of the state $q \in Q$ if there exists a path

$$\pi : q_0 \to^{\langle \alpha_1, \beta_1 \rangle} q_1 \to^{\langle \alpha_2, \beta_2 \rangle} q_2 \ldots \to^{\langle \alpha_n, \beta_n \rangle} q_n = q$$

starting from an initial state $q_0 \in I$ such that

$$\langle u, v \rangle = \omega^*(\langle \varepsilon, \varepsilon \rangle, \langle \alpha_1, \beta_1 \rangle \ldots \langle \alpha_n, \beta_n \rangle).$$

By $Adm(q)$ we denote the *set of all admissible advances* of the state $q \in Q$.

Corollary 4.6.7 *Let $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta \rangle$ be a classical real-time finite-state transducer. Let $\mathcal{S}_\mathcal{T}$ be the squared output transducer of $\mathcal{T}$, with set of final states $F' = F \times F$. Then $\mathcal{T}$ is functional iff for each $p \in F'$ we have $Adm(p) \subseteq \{\langle \varepsilon, \varepsilon \rangle\}$.*

Proof. Clearly, the admissible advances of the final states are exactly the advances of $\langle \varepsilon, \varepsilon \rangle$ with the labels of the successful paths in $\mathcal{S}_\mathcal{T}$. Part 2 of

Proposition 4.6.3 shows that an admissible advance of a final states is $\langle \varepsilon, \varepsilon \rangle$ iff for the label $\langle \alpha, \beta \rangle$ of the corresponding successful path in $\mathcal{S}_\mathcal{T}$ we have $\alpha = \beta$. Using Proposition 4.6.5 the result follows. □

Proposition 4.6.8 *Let $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta \rangle$ be a classical real-time finite-state transducer, let $\mathcal{S}_\mathcal{T}$ be the squared output transducer of $\mathcal{T}$, let p be a state on a successful path of $\mathcal{S}_\mathcal{T}$. Assume that*

1. *there exists $\langle u, v \rangle \in Adm(p)$ such that $\langle u, v \rangle$ is not balancible (i.e., $u \neq \varepsilon$ and $v \neq \varepsilon$), or*
2. *$|Adm(p)| > 1$.*

Then $\mathcal{T}$ is not functional.

Proof. Since p is on a successful path there exists a path π' starting at p and ending in a final state q of $\mathcal{S}_\mathcal{T}$. Let π' be such a path and $\langle \alpha', \beta' \rangle$ be the label of π'.

First, let $\langle u, v \rangle \in Adm(p)$ be not balancible. There exist a path π starting from a state $q_0 \in I \times I$ and ending in p with label $\langle \alpha, \beta \rangle$ such that $\omega(\langle \varepsilon, \varepsilon \rangle, \langle \alpha, \beta \rangle) = \langle u, v \rangle$. Then according to Proposition 4.6.3, Point 4 we have $\alpha\alpha' \neq \beta\beta'$. Since $\langle \alpha\alpha', \beta\beta' \rangle$ is the label of a successful path in $\mathcal{S}_\mathcal{T}$, from Proposition 4.6.5 it follows that $\mathcal{T}$ is not functional.

Now let $\langle u_1, v_1 \rangle, \langle u_2, v_2 \rangle \in Adm(p)$ be balancible admissible advances. If $\mathcal{T}$ is functional, then from Corollary 4.6.7 we obtain $\omega(\langle u_i, v_i \rangle, \langle \alpha', \beta' \rangle) = \langle \varepsilon, \varepsilon \rangle$ for $i = 1, 2$. From Proposition 4.6.3 Point 2 it follows that $u_i\alpha' = v_i\beta'$ and therefore $|u_i| - |v_i| = |\beta'| - |\alpha'|$ for $i = 1, 2$. Since $\langle u_1, v_1 \rangle$ and $\langle u_2, v_2 \rangle$ are balancible we assume without loss of generality that $v_1 = \varepsilon$, $|u_i| \geq |v_i|$ $(i = 1, 2)$ and therefore $v_2 = \varepsilon$. Hence $u_1\alpha' = u_2\alpha' = \beta'$ and therefore $u_1 = u_2$. □

Example 4.6.9 Consider the squared output transducer shown in Figure 4.6. The admissible advance of state $\langle 4, 6 \rangle$ contains $\langle c, d \rangle$, which is not balancible. Hence, if this state is on a successful path, then the underlying transducer cannot be functional.

The following two propositions provide a method for the effective construction of the function *Adm*.

Proposition 4.6.10 *Let $\mathcal{T}$ be a classical real-time finite-state transducer and let $\mathcal{S}_\mathcal{T} = \langle \Sigma^* \times \Sigma^*, Q \times Q, I \times I, F \times F, \Delta' \rangle$ be the squared output transducer of $\mathcal{T}$. Then the pair $\langle u, v \rangle \in \Sigma^* \times \Sigma^*$ is an admissible advance of $q \in Q \times Q$ iff*

- *$q \in I \times I$ and $\langle u, v \rangle = \langle \varepsilon, \varepsilon \rangle$, or*
- *there exist a state $q' \in Q \times Q$, an admissible advance $\langle u', v' \rangle$ of q' and a transition $\langle q', \langle \alpha, \beta \rangle, q \rangle \in \Delta'$ such that $\langle u, v \rangle = \omega(\langle u, v \rangle, \langle \alpha, \beta \rangle)$.*

Proof. Using Point 1 of Proposition 4.6.3, the proof follows directly from the observation that $\langle u, v \rangle \in Adm(q)$ iff there exists a path $\pi : q_0 \to^{\langle \alpha_1, \beta_1 \rangle} q_1 \to^{\langle \alpha_2, \beta_2 \rangle} q_2 \ldots \to^{\langle \alpha_n, \beta_n \rangle} q_n = q$ starting from $q_0 \in I \times I$ such that $\langle u, v \rangle = \omega^*(\langle \varepsilon, \varepsilon \rangle, \langle \alpha_1, \beta_1 \rangle \ldots \langle \alpha_n, \beta_n \rangle)$. □

Corollary 4.6.11 *Let $\mathcal{T}$ be a classical real-time finite-state transducer and let $\mathcal{S}_\mathcal{T} = \langle \Sigma^* \times \Sigma^*, Q \times Q, I \times I, F \times F, \Delta' \rangle$ be the squared output transducer of $\mathcal{T}$. Let the functions $Adm^{(k)} : Q \times Q \to 2^{\Sigma^* \times \Sigma^*}$ be defined inductively:*

1. $Adm^{(0)}(q) := \begin{cases} \{\langle \varepsilon, \varepsilon \rangle\} & \text{if } q \in I \times I \\ \emptyset & \text{otherwise.} \end{cases}$

2. $Adm^{(k+1)}(q) := Adm^{(k)}(q) \cup$
$\{\{\omega(\langle u', v' \rangle, \langle \alpha, \beta \rangle)\} \mid \langle q', \langle \alpha, \beta \rangle, q \rangle \in \Delta', \langle u', v' \rangle \in Adm^{(k)}(q')\}.$

Then $Adm = \bigcup_{k=0}^{\infty} Adm^{(k)}$.

General case – classical finite-state transducers

Corollary 4.6.12 *Let $\mathcal{T}$ be a classical finite-state transducer. Then T is functional iff*

1. $|(\{\varepsilon\} \times \Sigma^*) \cap L(\mathcal{T})| \leq 1$;
2. *$\mathcal{T}$ is not infinitely ambiguous;*
3. *$\mathcal{T}'$ is functional, where $\mathcal{T}'$ is the classical real-time finite-state transducer equivalent to $\mathcal{T}$ up to ε.*

Procedure for deciding functionality. The propositions above provide us with a procedure for deciding the functionality of a given classical finite-state transducer $\mathcal{T}$ with set of states Q. After checking functionality for empty input in a special subprocedure we convert $\mathcal{T}$ into a real-time transducer $\mathcal{T}'$ equivalent up to ε. The remaining procedure starts with building the corresponding trimmed squared output transducer $\mathcal{S}_{\mathcal{T}'}$ first and constructing the admissible advances for its states applying Corollary 4.6.11 afterwards. If during the construction we obtain two distinct admissible advances for a given state (which lies on a successful path) or if there is an admissible advance which is not balancible, or if an admissible advance of a final state is not equal to $\langle \varepsilon, \varepsilon \rangle$, then the construction terminates, $\mathcal{T}'$ is not functional. Otherwise at some step $k \leq |Q|^2$ the construction of the sets $Adm^{(k)}(q)$ stabilizes for all states $q \in Q \times Q$ and $\mathcal{T}'$ is functional. See Figure 4.7 for an illustration.

4.7 Summing up

We presented a brief introduction to n-tape automata, concentrating on closure properties. For proofs we used the fact that n-tape automata are special instances of monoidal finite-state automata. We have seen that the class of relations accepted by n-tape monoidal automata is closed under Cartesian product, projection and, for $n = 2$, under inversion of relations (Proposition 4.2.1). For classical n-tape automata there exist constructions which realize relational composition and reversal of relations (Proposition 4.3.5). A decision procedure for testing functionality of a classical transducer was described. Table 4.1 compares the general monoidal view with the special n-tape view.

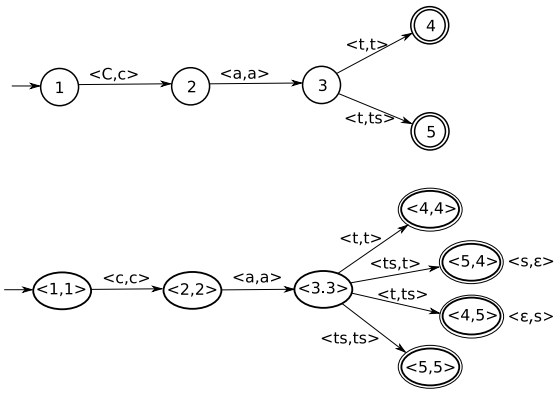

Figure 4.7: Illustration of the procedure for deciding functionality of a classical real-time finite-state transducer. At the bottom we see the squared output transducer for the input transducer on top. All states $\langle n, n\rangle$ have admissible advance $\langle \varepsilon, \varepsilon\rangle$. The final states $\langle 4, 5\rangle$ and $\langle 5, 4\rangle$ respectively have admissible advance $\langle \varepsilon, s\rangle$ and $\langle s, \varepsilon\rangle$ distinct from $\langle \varepsilon, \varepsilon\rangle$, which shows that the input transducer is not functional.

	General monoidal view	Special n-tape view
Device	Monoidal finite-state automaton	Monoidal n-type automaton
Languages recognized	Monoidal automaton language	Monoidal n-tape automaton relation
Closed under	Union, Concatenation, Kleene-Star	Union, Concatenation, Kleene-Star
	Homomorphic images	Homomorphic images,
	Cartesian product	Cartesian product, Projection
		Inverse relation for 2-tape automata
		Composition†, Reversal†
Equivalent lang. concept	Monoidal regular language	Monoidal regular n-relation
Translation device	Monoidal finite-state transducer	Classical finite-state transducer
		(free monoids, 2 tapes)

Table 4.1: Monoidal automata and the special case of monoidal n-tape automata. The symbol † indicates that the property holds for classical n-tape automata only. We assume that for translation devices the input tape always reads strings.

With this survey we finish the part where we study operations on languages and relations from a general point of view. In the following chapter we introduce a new focus, looking at *deterministic* devices.

4.8 Exercises for Chapter 4

Exercise 4.8.1 Let $\mathcal{N}$ denote the monoid of natural numbers with addition $+$ and neutral element 0, let $\mathcal{M} := \mathcal{N} \times \mathcal{N} \times \mathcal{N}$ denoite the monoid product with $n \geq 2$ factors $\mathcal{N}$. For given natural number coefficients $k_2, \ldots, k_n$ define an n-tape automaton $\mathcal{A}$ over M such that $L(\mathcal{A})$ is the set of all solutions of the equation $x_1 = \sum_{i=2,\ldots,n} k_i x_i$.

Exercise 4.8.2 The Levenshtein-distance $d_L(u, v)$ between two words $u, v \in \Sigma^*$ is the minimal number of letter insertions, letter deletions and letter substitutions needed to transform u into w. For a given alphabet Σ define a two-tape automaton $\mathcal{A}$ such that $L(\mathcal{A}) = \{\langle u, v\rangle \mid u, v \in \Sigma^*, d_L(u, v) \leq 1\}$ (cf. Example 8.2.13).

Exercise 4.8.3 Let $\Sigma := \{a, b\}$. For a string $w \in \Sigma^*$ let $w_{i(a)}$ denote the string where we delete in w the maximal prefix of the form a^n, let $w_{f(b)}$ denote the string where we delete in w the maximal suffix of the form b^m.

1. Describe a 2-tape letter automaton $\mathcal{A}_1$ such that
$$L(\mathcal{A}_1) := \{\langle w, w'\rangle \in \Sigma^* \times \Sigma^* \mid w_{i(a)} = w'_{i(a)}\}.$$
2. Describe a 2-tape letter automaton $\mathcal{A}_2$ such that
$$L(\mathcal{A}_2) := \{\langle w, w'\rangle \in \Sigma^* \times \Sigma^* \mid w_{f(b)} = w'_{f(b)}\}.$$
3. Use the construction introduced in the proof of Proposition 4.3.5 to construct a 2-tape automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{A}_1) \circ L(\mathcal{A}_2)$.

Exercise 4.8.4 Let $\mathcal{T}$ denote the monoidal letter transducer with set of states $\{1, 2, 3, 4, 5, 6\}$, single start state 1, single final state 6 and transitions

$$\begin{array}{lll} 1 & \to^{\langle \varepsilon, a\rangle} & 2 \\ 2 & \to^{\langle \varepsilon, b\rangle} & 3 \\ 2 & \to^{\langle \varepsilon, bb\rangle} & 4 \\ 3 & \to^{\langle a, bb\rangle} & 4 \\ 4 & \to^{\langle b, aa\rangle} & 3 \\ 4 & \to^{\langle \varepsilon, a\rangle} & 5 \\ 5 & \to^{\langle \varepsilon, b\rangle} & 6. \end{array}$$

Use the construction described in the proof of Proposition 4.4.8 to define an equivalent real-time transducer. Mark redundant transitions, if any.

Exercise 4.8.5 Let $L \subseteq \Sigma^*$ be a regular language. Show that $Id(L)$ is a binary regular string relation (cf. Example 4.5.3).

Exercise 4.8.6 Let Σ denote an alphabet. For $i = 1, \ldots, n$, let $\langle x_i, y_i \rangle \in \Sigma^* \times \Sigma^*$. Describe a simple way to compute

$$\omega^*(\langle x_1, y_1 \rangle, \langle \langle x_2, y_2 \rangle, \ldots, \langle x_n, y_n \rangle \rangle)$$

without using the recursive rule from Definition 4.6.1.

Exercise 4.8.7 Describe a simplified procedure for deciding functionality of classical finite-state transducers that works for all transducers that do not have any loop path.

Chapter 5

Deterministic transducers

Finite-state transducers as introduced in the previous chapter are non-deterministic rewriting devices. In this chapter we explore deterministic finite-state transducers. Obviously, it only makes sense to ask for determinism if we restrict attention to transducers with a functional input-output behaviour. In this chapter we focus on transducers that are deterministic on the input tape (called sequential or subsequential transducers). The theory of sequential finite-state transducers has been studied in [Eilenberg, 1974, Berstel, 1979, Sakarovitch, 2009]. We shall see that only a proper subset of all regular string functions can be represented by this kind of device. In the next chapter we introduce bimachines, a class of deterministic finite-state devices that exactly represents the class of all regular string functions.

5.1 Deterministic transducers and subsequential transducers

As discussed in Section 3.7 the notion of determinism is not natural for all monoidal automata. We now look at the special case of monoidal finite-state transducers. One natural notion of determinism is the following.

Definition 5.1.1 A monoidal finite-state transducer $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ is *deterministic* if the following conditions hold:

1. $|I| = 1$, i.e., there is exactly one initial state;
2. $\delta := \{\langle q_1, a, q_2 \rangle \mid \exists m \in M : \langle q_1, (a, m), q_2 \rangle \in \Delta\}$ is a (partial) function $Q \times \Sigma \to Q$;
3. $\lambda := \{\langle q_1, a, m \rangle \mid \exists q_2 \in Q : \langle q_1, (a, m), q_2 \rangle \in \Delta\}$ is a (partial) function $Q \times \Sigma \to M$;

The functions δ and λ are respectively called the *transition function* and the *transition output function*, and Δ is called the *transition relation*. Note that δ and λ have the same domain since both are derived from Δ. Deterministic

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

monoidal finite-state transducers are also denoted in the form

$$\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda \rangle$$

where $I = \{q_0\}$ and $\Delta = \{\langle q, (a, \lambda(q,a)), \delta(q,a)\rangle \mid \langle q, a\rangle \in dom(\delta)\}$.

Classical deterministic finite-state transducers are defined accordingly, demanding that the monoid $\mathcal{M}$ is free. Condition 2 says that when only looking at the first (string) tape, a deterministic transducer acts as a classical deterministic finite-state automaton. Condition 3 ensures that in each state the translation of the following alphabet symbol is unique. In the literature, distinct types of deterministic transducers were introduced and there is no generally accepted terminology, see Section 5.8.

Remark 5.1.2 Let $\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda \rangle$ be a deterministic monoidal finite-state transducer. The underlying finite-state automaton (cf. Def. 4.4.3) $\mathcal{A}_\mathcal{T} = \langle \Sigma, Q, q_0, F, \delta \rangle$ is a classical deterministic finite-state automaton such that $L(\mathcal{A}_\mathcal{T}) = L(\mathcal{T})_{\times 2}$. $L(\mathcal{A}_\mathcal{T})$ is called the *input language* of the transducer.

For a deterministic monoidal finite-state transducer $\mathcal{T}$, the *generalized transition function* δ^* is simply the generalized transition function of the underlying deterministic finite-state automaton. The *generalized transition output function* $\lambda^* : Q \times \Sigma^* \to \mathcal{M}$ of $\mathcal{T}$ has the same domain as δ^*. For all $q \in Q$ we define $\lambda^*(q, \varepsilon) := e$. Here e denotes the monoid unit element. Furthermore, if $q \in Q$, $t \in \Sigma^*$, $\sigma \in \Sigma$ and if $\delta^*(q, t\sigma)$ is defined we define $\lambda^*(q, t\sigma) := \lambda^*(q,t) \circ \lambda(\delta^*(q,t), \sigma)$. Here "$\circ$" denotes the monoid operation.

Remark 5.1.3 The monoidal language $L(\mathcal{T})$ recognized by a deterministic monoidal finite-state transducer $\mathcal{T}$ (cf. Definition 2.1.10) can be regarded as a function which maps words of the input language to an output monoid element. We call this function the *output function* $O_\mathcal{T} : L(\mathcal{T})_{\times 2} \to \mathcal{M}$ of the transducer. Alternatively the output function can be defined as $O_\mathcal{T}(t) = \lambda^*(q_0, t)$ for $t \in L(\mathcal{T})_{\times 2}$.

We now introduce another deterministic translation device. Its special feature is that final states come with their own output. As a simple motivation, consider the regular function $R := \langle a, c\rangle \cup \langle ab, d\rangle$. Since deterministic monoidal finite-state transducers at each transition read and translate a single input symbol, and since translation is compositional, no deterministic monoidal finite-state transducer can represent R: if a is translated into c, then the translation for ab cannot be d. Assume now we use a more general notion of transducer where final states may produce output strings when reached at the end of the input string. Then we may have a final state reached with input a with output c and another final state reached with input ab with output d. In this way, R can be represented without sacrifying determinism. The transducer is shown in Figure 5.1. Equivalently, we could add special $\langle \varepsilon, w\rangle$-transitions to new final states without outgoing transitions. However, this would destroy the deterministic behaviour of the transducer.

Definition 5.1.4 A *monoidal subsequential transducer* is a tuple

$$\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \Psi \rangle$$

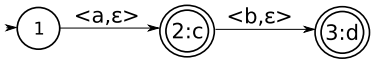

Figure 5.1: Subsequental transducer recognizing the regular function $R = \langle a, c\rangle \cup \langle ab, d\rangle$.

where $\langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda\rangle$ is a deterministic monoidal finite-state transducer and $\Psi : F \to \mathcal{M}$ is the *state output function* with domain F. The *underlying automaton* of $\mathcal{T}$ is the deterministic finite-state automaton $\mathcal{A}_\mathcal{T} = \langle \Sigma, Q, q_0, F, \delta\rangle$ and the *input language* of $\mathcal{T}$ is the set

$$L(\mathcal{T})_{\times 2} = L(\mathcal{A}_\mathcal{T}) = \{t \in \Sigma^* \mid \delta^*(q_0, t) \in F\}.$$

A *classical subsequential transducer* is a monoidal subsequential transducer where the monoid $\mathcal{M}$ is the free monoid Σ'^* over a finite *output alphabet* Σ'. Classical subsequential transducer are denoted also with $\langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \Psi\rangle$.

When considering classical subsequential transducers where the output monoid $\mathcal{M}$ is Σ^* we use the simpler notation $\langle \Sigma, Q, q_0, F, \delta, \lambda, \Psi\rangle$. The subsequential transducer reads words of the input language in a deterministic manner. Combining the output of the transitions and the output of the final state reached at the end, each accepted input word is mapped to a unique monoid element. The transducer thus represents a (partial) function.

Definition 5.1.5 Let $\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \Psi\rangle$ be a monoidal subsequential transducer. The *output function* $O_\mathcal{T} : L(\mathcal{T})_{\times 2} \to \mathcal{M}$ of $\mathcal{T}$ is defined as follows ("$\cdot$" represents the monoid operation):

$$\forall t \in L(\mathcal{T})_{\times 2} : \qquad O_\mathcal{T}(t) := \lambda^*(q_0, t) \cdot \Psi(\delta^*(q_0, t)).$$

Note that final states only produce an output when reached at the end of the input.

Definition 5.1.6 A monoidal deterministic or subsequential transducer $\mathcal{T}$ is said to *represent* the function $f : \Sigma^* \to \mathcal{M}$ iff $O_\mathcal{T} = f$.

Definition 5.1.7 Let $\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \Psi\rangle$ be a monoidal subsequential transducer. The *output function for* $q \in Q$ is

$$O^q_\mathcal{T} : \Sigma^* \to \mathcal{M};\ \alpha \mapsto \lambda^*(q, \alpha) \cdot \Psi(\delta^*(q, \alpha)).$$

The function is defined for $\alpha \in \Sigma^*$ iff $\delta^*(q, \alpha)$ is defined and in F.

Definition 5.1.8 Let $\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \Psi\rangle$ be a monoidal subsequential transducer. The states $q_1, q_2 \in Q$ are *equivalent* iff $O^{q_1}_\mathcal{T} = O^{q_2}_\mathcal{T}$.

We write $q_1 \equiv q_2$ to indicate that two states q_1 and q_2 are equivalent. When translating texts, devices are needed which produce an output for any input text. This leads to the following notion.

Definition 5.1.9 A *total monoidal subsequential transducer* is a subsequential transducer

$$\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \Psi\rangle$$

where $F = Q$ and δ and λ are total functions over $Q \times \Sigma$.

Direct composition of subsequential transducers

It is simple to give a direct construction for composition of subsequential transducers. For the following proposition and its proof, "$\cdot$" represents concatenation of strings, "$\circ$" denotes functional composition, and "$\odot$" denotes the monoid operation. An interesting point of the following construction is the definition of the set of final states of the new subsequential transducer, which takes into account that final outputs of the first transducer are translated by the second transducer in the composition. In the following definition of the transducer $\mathcal{T}$, the components δ, λ, F and Ψ only contain those entries where the function values used in the descriptions are defined.

Proposition 5.1.10 *Let $\mathcal{T}_1 = \langle \Sigma, \Sigma', Q_1, q_{01}, F_1, \delta_1, \lambda_1, \Psi_1 \rangle$ be a classical subsequential transducer and $\mathcal{T}_2 = \langle \Sigma', \mathcal{M}, Q_2, q_{02}, F_2, \delta_2, \lambda_2, \Psi_2 \rangle$ be a monoidal subsequential transducer. Then for the transducer*

$$\mathcal{T} = \langle \Sigma, \mathcal{M}, Q_1 \times Q_2, \langle q_{01}, q_{02} \rangle, F, \delta, \lambda, \Psi \rangle$$

where

- $\delta = \{\langle \langle q_1, q_2 \rangle, \sigma, \langle \delta_1(q_1, \sigma), \delta_2^*(q_2, \lambda_1(q_1, \sigma)) \rangle \rangle \mid q_1 \in Q_1, q_2 \in Q_2, \sigma \in \Sigma\}$,
- $\lambda = \{\langle \langle q_1, q_2 \rangle, \sigma, \lambda_2^*(q_2, \lambda_1(q_1, \sigma)) \rangle \mid q_1 \in Q_1, q_2 \in Q_2, \sigma \in \Sigma\}$,
- $F = \{\langle q_1, q_2 \rangle \mid q_1 \in F_1, \delta_2^*(q_2, \Psi_1(q_1)) \in F_2\}$,
- $\Psi = \{\langle \langle q_1, q_2 \rangle, \lambda_2^*(q_2, \Psi_1(q_1)) \odot \Psi_2(\delta_2^*(q_2, \Psi_1(q_1))) \rangle \mid \langle q_1, q_2 \rangle \in F\}$;

we have $O_{\mathcal{T}} = O_{\mathcal{T}_1} \circ O_{\mathcal{T}_2}$.

Proof. "$\subseteq$": Let $\alpha \in dom(O_{\mathcal{T}_1} \circ O_{\mathcal{T}_2})$ and $(O_{\mathcal{T}_1} \circ O_{\mathcal{T}_2})(\alpha) = O_{\mathcal{T}_2}(O_{\mathcal{T}_1}(\alpha)) = m$. Let

$$\beta = O_{\mathcal{T}_1}(\alpha) = \lambda_1^*(q_{01}, \alpha) \cdot \Psi_1(\delta_1^*(q_{01}, \alpha)),$$

let $q_1 := \delta_1^*(q_{01}, \alpha)$ and $q_2 := \delta_2^*(q_{02}, \lambda_1^*(q_{01}, \alpha))$. Then

$$\begin{aligned} m &= \lambda_2^*(q_{02}, \beta) \odot \Psi_2(\delta_2^*(q_{02}, \beta)) \\ &= \lambda_2^*(q_{02}, \beta) \odot \lambda_2^*(\Psi_1(q_1)) \odot \Psi_2(\delta_2^*(q_2, \Psi_1(q_1))). \end{aligned}$$

Following the definition of $\mathcal{T}$ it is easy to see that $\delta^*(\langle q_{01}, q_{02} \rangle, \alpha) = \langle q_1, q_2 \rangle \in F$ and

$$m = \lambda^*(\langle q_{01}, q_{02} \rangle, \alpha) \odot \Psi(\delta^*(\langle q_{01}, q_{02} \rangle, \alpha)).$$

"$\supseteq$": Let $\alpha \in dom(O_{\mathcal{T}})$. Since $\delta^*(\langle q_{01}, q_{02} \rangle, \alpha) = \langle \delta_1^*(q_{01}, \alpha), \delta_2^*(q_{02}, \lambda_1^*(q_{01}, \alpha)) \rangle$, we have $q_1 = \delta_1^*(q_{01}, \alpha) \in F_1$ and therefore $\alpha \in dom(O_{\mathcal{T}_1})$. Moreover we have $\delta_2^*(q_2, \Psi_1(q_1)) \in F_2$ where $q_2 = \delta_2^*(q_{02}, \lambda_1^*(q_{01}, \alpha))$.
Therefore for $\beta = O_{\mathcal{T}_1}(\alpha) = \lambda_1^*(q_{01}, \alpha) \cdot \Psi_1(\delta_1^*(q_{01}, \alpha))$ we have $\beta \in dom(O_{\mathcal{T}_2})$ and

$$\begin{aligned} O_{\mathcal{T}}(\alpha) &= \lambda^*(\langle q_{01}, q_{02} \rangle, \alpha) \odot \Psi(\delta^*(\langle q_{01}, q_{02} \rangle, \alpha)) \\ &= \lambda_2^*(q_{02}, \beta) \odot \lambda_2^*(\Psi_1(q_1)) \odot \Psi_2(\delta_2^*(q_2, \Psi_1(q_1))) \\ &= \lambda_2^*(q_{02}, \beta) \odot \Psi_2(\delta_2^*(q_{02}, \beta)) \\ &= (O_{\mathcal{T}_1} \circ O_{\mathcal{T}_2})(\alpha) = O_{\mathcal{T}_2}(O_{\mathcal{T}_1}(\alpha)) \end{aligned}$$

□

The bounded variation property

Remark 5.1.11 A string function $f : \Sigma^* \to \Sigma'^*$ is called a *subsequential function* (cf. [Roche and Schabes, 1997b]) if it can be represented by a classical subsequential transducer. Choffrut [Choffrut, 1977] has shown that it is decidable if a string function is subsequential. There exist effective determinization procedures that transfer a given classical finite-state transducer representing a subsequential string function f into an equivalent classical subsequential transducer [Roche and Schabes, 1997a, Mohri, 1996]. Below we present such a determinization construction. However, not all regular string functions can be represented by means of classical subsequential transducers. As an example consider the regular function $R := \langle a, a\rangle^* \cup (\langle a, \varepsilon\rangle^* \cdot \langle b, b\rangle)$. Here a sequence of letters a is copied, whereas a sequence of letters a which is followed by a single letter b is translated into b. In order to represent this function the device has to delay the output until the last symbol is read. Here the device has to reconstruct eventually the number of letters a read. Since this number is not bounded, such a behaviour cannot be realized using a deterministic *finite*-state transducer. In the next section we show that a regular string function $f : \Sigma^* \to \Sigma'^*$ can be represented by a classical subsequential transducer iff f has the so-called "bounded variation property", which is defined below.

Recall that $u \wedge v$ denotes the longest common prefix of the two strings u and v.

Definition 5.1.12 The *sequential distance* of $u \in \Sigma^*$ and $v \in \Sigma^*$ is defined as $d_S(u, v) = |u| + |v| - 2|u \wedge v|$.

Definition 5.1.13 A regular string function $f : \Sigma^* \to \Sigma'^*$ has the *bounded variation property* iff for all $k \geq 0$ there exists $K \geq 0$ such that for all $u, v \in dom(f)$ always $d_S(u, v) \leq k$ implies $d_S(f(u), f(v)) \leq K$.

The definition roughly says that two input strings that are identical up to small suffixes are translated into output strings that are similar up to suffixes.

Definition 5.1.14 A functional classical transducer $\mathcal{T}$ has the *bounded variation property* iff the regular string function represented by $\mathcal{T}$ has the bounded variation property.

When looking at constructions for subsequential transducers one annoying technical detail is often caused by the output of the empty input string. To avoid these problems, in the following parts we only look at subsequential transducers where the empty word does not belong to the input language. The following lemma gives a justification. It shows how to add an arbitrary mapping for the empty word to the output function of a subsequential transducer.

Lemma 5.1.15 *Let $\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \Psi\rangle$ be a monoidal subsequential transducer such that $q_0 \notin F$. Let $m_0 \in M$ be a monoid element. Let q_0' be a new state such that $q_0' \notin Q$. Let*

$$\mathcal{T}' := \langle \Sigma, \mathcal{M}, Q \cup \{q_0'\}, q_0', F \cup \{q_0'\}, \delta', \lambda', \Psi \cup \{\langle q_0', m_0\rangle\}\rangle$$

where $\delta' = \delta \cup \{\langle q_0', a, p\rangle \mid \langle q_0, a, p\rangle \in \delta\}$ and $\lambda' = \lambda \cup \{\langle q_0', a, m\rangle \mid \langle q_0, a, m\rangle \in \lambda\}$. Then we have

$$O_{\mathcal{T}'} = O_{\mathcal{T}} \cup \{\langle \varepsilon, m_0\rangle\}.$$

Proof. Clearly by cloning the initial state we do not change the output function of the transducer. The new initial state has no incoming transitions. Therefore setting its output to m_0 will add only the mapping $\langle \varepsilon, m_0 \rangle$ to the output function of $\mathcal{T}'$. □

5.2 A determinization procedure for functional transducers with the bounded variation property

In this part it is shown that any functional classical transducer with the bounded variation property can be transformed into an equivalent classical *subsequential* transducer. We will follow the determinization procedure described in [Roche and Schabes, 1997b]. In Section 5.6 we later show how the results from this and the following sections can be generalized for other monoids.

We assume that a functional classical transducer $\mathcal{T}$ over $\Sigma \times \Sigma'^*$ is given. For simplicity we assume that $\varepsilon \notin L(\mathcal{T})_{\times 2}$. This assumption is not significant, cf. Lemma 5.1.15. Hence we may assume that the given input transducer

$$\mathcal{T} = \langle \Sigma^* \times \Sigma'^*, Q, I, F, \Delta \rangle$$

is real-time, which implies that $\Delta \subseteq Q \times (\Sigma \times \Sigma'^*) \times Q$. We may also assume that each state $q \in Q$ is on a successful path of $\mathcal{T}$.

The following iterative construction can be used for any source transducer satisfying the above conditions. However, for ensuring termination it is necessary that $\mathcal{T}$ has the bounded variation property. Hence at a certain point below we shall assume that $\mathcal{T}$ has the bounded variation property. Obviously, the translation into a real-time transducer (cf. Proposition 4.4.8) again leads to a functional transducer with the bounded variation property.

The subsequential transducer

$$\mathcal{T}' = \langle \Sigma, \Sigma', Q', q'_0, F', \delta', \lambda', \Psi' \rangle$$

equivalent to $\mathcal{T}$ is built by induction. At each step we extend the current set of states Q', the set of final states F', the transition function δ' and the output functions λ' and Ψ'.

The construction can be seen as a special power set construction. Each state of Q' keeps track of a set of states reached in the source transducer $\mathcal{T}$ with the parallel input on distinct paths. In $\mathcal{T}$, parallel paths for the same input may have distinct outputs. In $\mathcal{T}'$ we only produce the maximal common output prefix of all such parallel runs. This means that we need to store with each state $p \in Q$ belonging to a state $S \in Q'$ the delayed output. Hence entries of states $S \in Q'$ are pairs of the form $\langle p, u \rangle$ where $p \in Q$ and $u \in \Sigma'^*$ represents the delayed output at p. When reaching a final state, delayed output is finally produced as state output. We use the following notation. If $S \subseteq \Sigma^*$ is a non-empty set of strings, by $\bigwedge_{v \in S} v$ we denote the maximal common prefix of all strings in S (see Definition 1.3.5).

The base of the induction is:

$$\mathcal{T}'^{(0)} = \langle \Sigma, \Sigma', \{q'_0\}, q'_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle \text{ , where } q'_0 = I \times \{\varepsilon\}.$$

This transducer only represents runs with empty input in the source transducer $\mathcal{T}$, the delayed output of any state in I is ε. Let us assume that we have constructed

$$\mathcal{T}'^{(n)} = \left\langle \Sigma, \Sigma', Q'^{(n)}, q'_0, F'^{(n)}, \delta'^{(n)}, \lambda'^{(n)}, \Psi'^{(n)} \right\rangle.$$

We define $\mathcal{T}'^{(n+1)} = \langle \Sigma, \Sigma', Q'^{(n+1)}, q'_0, F'^{(n+1)}, \delta'^{(n+1)}, \lambda'^{(n+1)}, \Psi'^{(n+1)} \rangle$ with the following components:

- The new transition output function $\lambda'^{(n+1)}$ extends $\lambda'^{(n)}$ with all triples of the form $\langle S, \sigma, w \rangle$ for which $\{\langle q, \langle \sigma, v \rangle, q' \rangle \in \Delta \mid \langle q, u \rangle \in S\} \neq \emptyset$, where $S \in Q'^{(n)}$, $\sigma \in \Sigma$ and

$$w = \bigwedge_{\langle q,u\rangle \in S} \bigwedge_{\langle q,\langle \sigma,v\rangle,q'\rangle \in \Delta} u \cdot v.$$

  Here w represents the maximal common prefix of all strings obtained by concatenating a delayed output u in S with the new transition output v.

- The new transition function $\delta'^{(n+1)}$ extends $\delta'^{(n)}$ with all triples of the form $\langle S, \sigma, S' \rangle$ where $\langle S, \sigma, w \rangle$ is a triple in $\lambda'^{(n+1)}$ and

$$S' = \bigcup_{\langle q,u\rangle \in S} \bigcup_{\langle q,\langle \sigma,v\rangle,q'\rangle \in \Delta} \{\langle q', w^{-1}(u \cdot v)\rangle\}$$

  Here we use the notation $^{-1}$ introduced in Definition 1.3.4, the string $w^{-1}(u \cdot v)$ represents the new delayed output after the transition with σ.

- $Q'^{(n+1)} = Q'^{(n)} \cup codom(\delta'^{(n+1)})$,
- $F'^{(n+1)} = \{S \in Q'^{(n+1)} \mid \exists \langle q, \beta \rangle \in S \ : \ q \in F\}$,
- $\Psi'^{(n+1)} = \{\langle S, \beta \rangle \mid S \in F'^{(n+1)}, \exists \langle q, \beta \rangle \in S \ : \ q \in F\}$.

At each step the transducer is extended with a finite number of states reachable with one symbol from the set of states of the current transducer. It is simple to see that at each step n always $\lambda'^{(n)}$ and $\delta'^{(n)}$ are in fact proper *functions*, which means that the extension steps do not lead to multiple values.

(†) The generalized versions of these functions are defined for all $w \in \Sigma^*$ such that $\langle i, w, q \rangle \in \Delta^*$ for some $i \in I$ and $q \in Q$.

We shall see below (Lemma 5.2.4) that $\Psi'^{(n)}$ is always well-defined. Hence after each step we obtain a proper classical subsequential transducer.

Example 5.2.1 The determinization construction is illustrated in Figure 5.2. The transducer in the upper part is functional, representing the output function $\{\langle a, cc \rangle, \langle ab, cccdd \rangle, \langle aba, cccdd \rangle\}$. In the lower part we see the states of the determinized transducer, where states with delayed input are combined. The state output of the three resulting final states is respectively given by the delayed output c, d, and ε for the final states 4, 7, and 8.

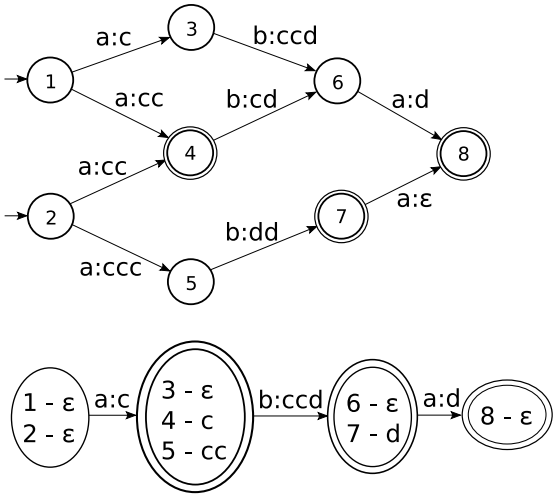

Figure 5.2: Illustration for determinization of functional transducers, cf. Example 5.2.1.

With the help of the following lemmas[1] we will show that $O_{\mathcal{T}'} = L(\mathcal{T})$, which means that the new transducer is equivalent to the source transducer $\mathcal{T}$.

Lemma 5.2.2 *Let $\mathcal{T}$ be as above. Let $\mathcal{T}'^{(n)} = \langle \Sigma, \Sigma', Q', q_0', F', \delta', \lambda', \Psi' \rangle$ be constructed by the induction given above in n steps. Then for each word $w \in \Sigma^*$ such that $\lambda'^*(q_0', w)$ and $\delta'^*(q_0', w)$ are defined we have the following properties:*

$$\begin{aligned} \lambda'^*(q_0', w) &= \bigwedge_{q_0 \in I, \langle q_0, \langle w, u \rangle, q \rangle \in \Delta^*} u \\ \delta'^*(q_0', w) &= \{ \langle q, \gamma \rangle \mid \exists q_0 \in I \; \exists \langle q_0, \langle w, u \rangle, q \rangle \in \Delta^* \; : \; \gamma = \lambda'^*(q_0', w)^{-1} u \}. \end{aligned}$$

Proof. The proof proceeds by induction on the length of w. For $|w| = 0$ the above equations are obviously true since $\Delta^* \cap I \times (\{\varepsilon\} \times \Sigma'^*) \times Q = \{ \langle q_0, \langle \varepsilon, \varepsilon \rangle, q_0 \rangle \mid q_0 \in I \}$. Let us assume that the above equations are true for $w \in \Sigma^*$. We show that the equations hold for $w' = wa$, for all $a \in \Sigma$ such that $\delta'^*(q_0', wa)$ is defined:

$$\begin{aligned} \lambda'^*(q_0', wa) &= \lambda'^*(q_0', w) \; \lambda'(\delta'^*(q_0', w), a) \\ &\overset{\text{Def. } \lambda'}{=} \lambda'^*(q_0', w) \bigwedge_{\langle q, v \rangle \in \delta'^*(q_0', w)} \bigwedge_{\langle q, \langle a, v' \rangle, q' \rangle \in \Delta} vv' \\ &\overset{\text{Ind. hyp.}}{=} \lambda'^*(q_0', w) \bigwedge_{q_0 \in I, \langle q_0, \langle w, u \rangle, q \rangle \in \Delta^*} \bigwedge_{\langle q, \langle a, v' \rangle, q' \rangle \in \Delta} \lambda'^*(q_0', w)^{-1} uv' \\ &= \bigwedge_{q_0 \in I, \langle q_0, \langle w, u \rangle, q \rangle \in \Delta^*} \bigwedge_{\langle q, \langle a, v' \rangle, q' \rangle \in \Delta} uv' \\ &= \bigwedge_{q_0 \in I, \langle q_0, \langle wa, u' \rangle, q' \rangle \in \Delta^*} u'. \end{aligned}$$

[1] The lemmas can be found in a similar form in [Roche and Schabes, 1997b].

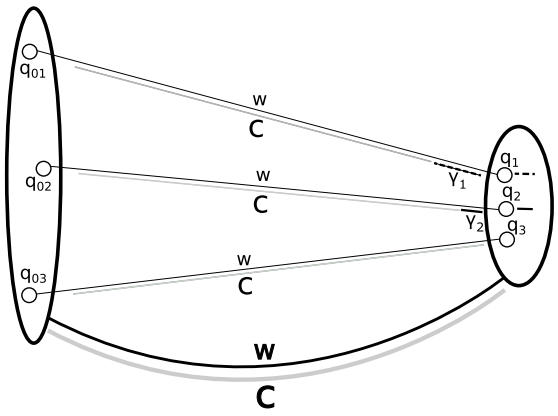

Figure 5.3: Illustration for Lemma 5.2.2. Straight lines indicate three paths of the source transducer with the same input label w respectively starting from the three initial states q_{01}, q_{02}, and q_{03} and leading to q_1, q_2, and q_3. The maximal common output $C = \lambda'^*(q'_0, w)$ of all paths is shown as a grey line. On the first two paths, additional outputs γ_1 and γ_2 are produced. States and transitions of the target transducer are shown with bold lines. For input w, producing output C a complex state $\{\langle q_1, \gamma_1\rangle, \langle q_2, \gamma_2\rangle, \langle q_3, \varepsilon\rangle\}$ is reached.

This proves the first equation. Furthermore we have

$$\begin{aligned}
\delta'^*(q'_0, wa) &= \delta'(\delta'^*(q'_0, w), a) \\
&= \bigcup_{\langle q,v\rangle \in \delta'^*(q'_0,w)} \; \bigcup_{\langle q,\langle a,v'\rangle,q'\rangle \in \Delta} \{\langle q', \lambda'(\delta'^*(q'_0, w), a)^{-1}\ vv'\rangle\} \\
&= \bigcup_{q_0 \in I, \langle q_0,\langle w,u\rangle,q\rangle \in \Delta^*} \; \bigcup_{\langle q,\langle a,v'\rangle,q'\rangle \in \Delta} \{\langle q', \lambda'(\delta'^*(q'_0, w), a)^{-1}\ \lambda'^*(q'_0, w)^{-1} uv'\rangle\} \\
&= \bigcup_{q_0 \in I, \langle q_0,\langle wa,u'\rangle,q'\rangle \in \Delta^*} \{\langle q', \lambda'^*(q'_0, wa)^{-1} u'\rangle\} \\
&= \{\langle q', v''\rangle \mid \exists q_0 \in I\ \exists \langle q_0, \langle w, u'\rangle, q'\rangle \in \Delta^*\ :\ v'' = \lambda'^*(q'_0, wa)^{-1} u'\}.
\end{aligned}$$

This concludes the proof of the second equation. □

Lemma 5.2.2 is illustrated in Figure 5.3. Note that in general a state $\delta'^*(q'_0, w)$ of the new transducer can be reached with distinct input words w'. The above equations hold for each such w'.

Lemma 5.2.3 *Let $\mathcal{T}'^{(n)} = \langle \Sigma, \Sigma', Q', q'_0, F', \delta', \lambda', \Psi'\rangle$ be constructed by the induction given above in n steps from the transducer $\mathcal{T} = \langle \Sigma^* \times \Sigma'^*, Q, \{q_0\}, F, \Delta\rangle$. Then for each state $S \in Q'$ and $q \in Q$ we have $|\{v \in \Sigma'^* \mid \exists \langle q, v\rangle \in S\}| \leq 1$.*

Proof. Let us assume that there exist $\langle q, v_1\rangle, \langle q, v_2\rangle \in S$. Let $w \in \Sigma^*$ be any word such that $S = \delta'^*(q'_0, w)$. Part 2 of the previous lemma shows that there exist $u_1, u_2 \in \Sigma^*$ and $q_{01}, q_{02} \in I$ such that

$$\begin{aligned}
&\langle q_{01}, \langle w, u_1\rangle, q\rangle \in \Delta^*, \qquad v_1 = \lambda'^*(q'_0, w)^{-1} u_1 \\
&\langle q_{02}, \langle w, u_2\rangle, q\rangle \in \Delta^*, \qquad v_2 = \lambda'^*(q'_0, w)^{-1} u_2
\end{aligned}$$

Since the source transducer $\mathcal{T}$ is functional and each state q lies on a successful path the left-hand side relations show that $u_1 = u_2$. Now the right-hand side equations show that $v_1 = v_2$. □

Lemma 5.2.4 *Let $\mathcal{T}'^{(n)} = \langle \Sigma, \Sigma', Q', q'_0, F', \delta', \lambda', \Psi' \rangle$ be constructed by the induction given above in n steps from the transducer $\mathcal{T} = \langle \Sigma^* \times \Sigma'^*, Q, \{q_0\}, F, \Delta \rangle$. Then for each state $S \in Q'$ and $\langle q_1, v_1 \rangle, \langle q_2, v_2 \rangle \in S$ we have*

$$q_1 \in F \;\&\; q_2 \in F \to v_1 = v_2.$$

Proof. Let us assume that $\langle q_1, v_1 \rangle, \langle q_2, v_2 \rangle \in S$. Let $w \in \Sigma^*$ be any word such that $S = \delta'^*(q'_0, w)$. The above lemma shows that there exist $u_1, u_2 \in \Sigma'^*$ and $q_{01}, q_{02} \in I$ such that

$$\begin{aligned} \langle q_{01}, \langle w, u_1 \rangle, q_1 \rangle \in \Delta^*, \qquad & v_1 = \lambda'^*(q'_0, w)^{-1} u_1 \\ \langle q_{02}, \langle w, u_2 \rangle, q_2 \rangle \in \Delta^*, \qquad & v_2 = \lambda'^*(q'_0, w)^{-1} u_2 \end{aligned}$$

Since the source transducer $\mathcal{T}$ is functional and $q_1, q_2 \in F$ the left-hand side relations show that $u_1 = u_2$. The right-hand side equations imply $v_1 = v_2$. □

Proposition 5.2.5 *Let $\mathcal{T} = \langle \Sigma^* \times \Sigma'^*, Q, I, F, \Delta \rangle$ be a functional real-time classical transducer such that the inductive construction of $\mathcal{T}'$ presented above terminates in the sense that there exists a number $k \in \mathbb{N}$ such that $\mathcal{T}'^{(k)} = \mathcal{T}'^{(k+1)} = \mathcal{T}'^{(k+2)} = \ldots = \mathcal{T}'$, let $\mathcal{T}' = \langle \Sigma, \Sigma', Q', q'_0, F', \delta', \lambda', \Psi' \rangle$. Then $O_{\mathcal{T}'} = L(\mathcal{T})$.*

Proof. ("$\subseteq$") Let $w \in \Sigma^*$, assume that $O_{\mathcal{T}'}(w)$ is defined. Then we have $O_{\mathcal{T}'}(w) = \lambda'^*(q'_0, w)\Psi'(q')$, where $q' := \delta'^*(q'_0, w) \in F'$. Lemma 5.2.2 shows that

$$q' = \{\langle q, v' \rangle \mid \exists q_0 \in I \; \exists \langle q_0, \langle w, u' \rangle, q \rangle \in \Delta^* : v' = \lambda'^*(q'_0, w)^{-1} u'\},$$

and by definition of F' there exists $\langle q, v \rangle \in q'$ such that $q \in F$ and $\Psi'(q') = v$. Hence there exist $q_0 \in I, \langle q_0, \langle w, u \rangle, q \rangle \in \Delta^*$ such that $v = \lambda'^*(q'_0, w)^{-1} u$. It follows that

$$O_{\mathcal{T}'}(w) = \lambda'^*(q'_0, w)\lambda'^*(q'_0, w)^{-1} u = u.$$

Since $q_0 \in I$ and $q \in F$ we also have $\langle w, u \rangle \in L(\mathcal{T})$. Hence $O_{\mathcal{T}'} \subseteq L(\mathcal{T})$.

("$\supseteq$") Assume now that $\langle q_0, \langle w, u \rangle, q \rangle \in \Delta^*$ and $q_0 \in I, q \in F$. Since $\mathcal{T}'^{(k)} = \mathcal{T}'^{(k+1)}$ the above observation (†) shows that δ'^* and λ'^* are defined for w. The construction of $\mathcal{T}'$ implies that $\lambda'^*(q'_0, w)$ is a prefix of $\lambda^*(w) = u$. Let $v := \lambda'^*(q'_0, w)^{-1} u$. Then $\langle q, v \rangle$ occurs in

$$\delta'^*(q'_0, w) = \{\langle q, v' \rangle \mid \exists q_0 \in I \; \exists \langle q_0, \langle w, u' \rangle, q \rangle \in \Delta^* : v' = \lambda'^*(q'_0, w)^{-1} u'\}.$$

Since $q \in F$ we have $\delta'^*(q'_0, w) \in F'$ and $\Psi'(\delta'^*(q'_0, w)) = v$. Hence

$$O_{\mathcal{T}'}(w) = \lambda'^*(q'_0, w)v = \lambda'^*(q'_0, w)\lambda'^*(q'_0, w)^{-1} u = u$$

and $L(\mathcal{T}) \subseteq O_{\mathcal{T}'}$. □

Theorem 5.2.6 *Let $\mathcal{T} = \langle \Sigma^* \times \Sigma'^*, Q, I, F, \Delta \rangle$ be a trimmed real-time functional classical transducer with the bounded variation property. Then the inductive construction of $\mathcal{T}'$ presented above terminates in the sense that there exists a number $k \in \mathbb{N}$ such that $\mathcal{T}'^{(k)} = \mathcal{T}'^{(k+1)} = \mathcal{T}'^{(k+2)} = \ldots = \mathcal{T}'$.*

Proof. Let us choose for each state $q \in Q$ a path

$$\pi_q : q \to \ldots^{\langle w_q, \alpha_q \rangle} \to t_q$$

with label $w_q \in \Sigma^*$ to a final state $t_q \in F$. Since $\mathcal{T}$ is trimmed such a path exists for each q. Let $m = \max_{q \in Q} |w_q|$ be the longest input label of all paths π_q. Let $D = \max_{q \in Q} |\alpha_q|$ denote the maximum length of the outputs produced on the paths π_q for $q \in Q$. Since $\mathcal{T}$ has the bounded variation property, for $k = 2m$ there exists $K \in \mathbb{N}$ such that $d_S(u, v) \leq k$ implies $d_S(f(u), f(v)) \leq K$. Let us assume that the inductive construction does not terminate. In this case the sequence of sets $Q'^{(n)}$ is strongly monotonically growing. Since $Q'^{(n)} \subseteq 2^{Q \times \Sigma'^*}$ and the number of subsets of states in Q is $2^{|Q|}$ there exist an $n \in \mathbb{N}$ and a set $S \in Q'^{(n)}$ containing a pair $\langle q_1, \gamma_1 \rangle \in S$ such that $|\gamma_1| > K + D$. From the inductive construction it follows that there exists another pair $\langle q_2, \gamma_2 \rangle \in S$ such that $\gamma_1 \wedge \gamma_2 = \varepsilon$ $(*)$. Let S be reachable in $\mathcal{T}'^{(n)}$ with the word $u \in \Sigma^*$, i.e. $\delta'^{(n)^*}(q'_0, u) = S$. In this case there exist two successful paths in $\mathcal{T}$

$$\begin{aligned} \pi_1 &= q_{01} \to \ldots^{\langle u, \lambda'^{(n)^*}(q'_0, u)\gamma_1 \rangle} \to q_1 \to \ldots^{\langle w_{q_1}, \alpha_{q_1} \rangle} \to t_{q_1} \\ \pi_2 &= q_{02} \to \ldots^{\langle u, \lambda'^{(n)^*}(q'_0, u)\gamma_2 \rangle} \to q_2 \to \ldots^{\langle w_{q_2}, \alpha_{q_2} \rangle} \to t_{q_2}. \end{aligned}$$

Clearly $d_S(uw_{q_1}, uw_{q_2}) \leq |w_{q_1}| + |w_{q_2}| \leq 2m = k$. For the images of the two words we consider two cases.
Case 1: $\gamma_2 \neq \varepsilon$, In that case we obtain

$$\begin{aligned} d_S(L(\mathcal{T})(uw_{q_1}), L(\mathcal{T})(uw_{q_2})) &= d_S(\lambda'^{(n)^*}(q'_0, u)\gamma_1\alpha_{q_1}, \lambda'^{(n)^*}(q'_0, u)\gamma_2\alpha_{q_2}) \\ &\overset{(*)}{=} |\gamma_1| + |\gamma_2| + |\alpha_{q_1}| + |\alpha_{q_2}| > |\gamma_1| > K. \end{aligned}$$

Case 2: $\gamma_2 = \varepsilon$. In that case $|\gamma_1\alpha_{q_1}| > K + D$ and $|\alpha_{q_2}| < D$ $(**)$

$$\begin{aligned} d_S(L(\mathcal{T})(uw_{q_1}), L(\mathcal{T})(uw_{q_2})) &= d_S(\lambda'^{(n)^*}(q'_0, u)\gamma_1\alpha_{q_1}, \lambda'^{(n)^*}(q'_0, u)\gamma_2\alpha_{q_2}) \\ &= d_S(\gamma_1\alpha_{q_1}, \alpha_{q_2}) \overset{(**)}{>} (K + D) - D = K. \end{aligned}$$

This contradiction shows that our assumption must be wrong and the inductive construction terminates. □

From this theorem it follows that $\mathcal{T}'$ has a finite number of states and that δ' and λ' are proper finite functions.

Corollary 5.2.7 *A regular string function $f : \Sigma^* \to \Sigma'^*$ can be represented by a classical subsequential transducer iff f has the bounded variation property.*

Proof. Let $f : \Sigma^* \to \Sigma'^*$ be regular.
("⇐") Since f is regular it can be represented by a classical functional transducer (Theorem 4.5.5). It follows (cf. Section 2.5, Proposition 3.7.2) that there

exists a trimmed pseudo-deterministic functional classical transducer $\mathcal{T}$ representing f. If f has the bounded variation property, then the inductive transducer construction terminates (Theorem 5.2.6). It follows from Lemma 5.2.4 that the construction gives a classical subsequential transducer. This transducer represents f (Proposition 5.2.5).
("⇒") To prove the other direction, let f be represented as a subsequential transducer $\mathcal{T} = \langle \Sigma, \Sigma'^*, Q, q_0, F, \delta, \lambda, \Psi \rangle$. Let $C = \max_{\langle q,a,\alpha\rangle \in \lambda} |\alpha|$ and $D = \max_{\langle q,\alpha\rangle \in \Psi} |\alpha|$. Let $k \geq 0$ be arbitrary. Let us choose $K = Ck + 2D$. Let $u, v \in dom(O_{\mathcal{T}})$, $w = u \wedge v$ and $u = wu', v = wv'$, such that $d_S(u, v) \leq k$. Then we have two successful paths in $\mathcal{T}$

$$\begin{aligned} \pi_1 &= q_0 \to \ldots^{\langle w, \lambda^*(q_0,w)\rangle} \to q' \to \ldots^{\langle u', \lambda^*(q',u')\rangle} \to t_1 \\ \pi_2 &= q_0 \to \ldots^{\langle w, \lambda^*(q_0,w)\rangle} \to q' \to \ldots^{\langle v', \lambda^*(q',v')\rangle} \to t_2, \end{aligned}$$

and

$$\begin{aligned} O_{\mathcal{T}}(u) &= \lambda^*(q_0, w)\lambda^*(q', u')\Psi(t_1) \\ O_{\mathcal{T}}(v) &= \lambda^*(q_0, w)\lambda^*(q', v')\Psi(t_2). \end{aligned}$$

We obtain

$$\begin{aligned} d_S(O_{\mathcal{T}}(u), O_{\mathcal{T}}(v)) &= d_S(\lambda^*(q', u')\Psi(t_1), \lambda^*(q_0, w)\lambda^*(q', v')) \\ &\leq |\lambda^*(q', u')| + |\lambda^*(q', v')| + |\Psi(t_1)| + |\Psi(t_2)| \\ &\leq (|u'| + |v'|)C + 2D = d_S(u, v)C + 2D \leq K. \end{aligned}$$

□

5.3 Deciding the bounded variation property

In order to see if we can successfully apply the transducer determinization procedure presented in Section 5.2 to a functional classical transducer we need to decide if the transducer has the bounded variation property. In this section we present a decision algorithm based on the approach in [Béal et al., 2003].

The first lemma provides the key tools used later in the decision procedure. We use the notation introduced in Section 4.6. In particular ω^* denotes the iterated advance function introduced in Definition 4.6.1.

Lemma 5.3.1 *Let $z, u, v \in \Sigma^*$ and $uv \neq \varepsilon$. For $n \in \mathbb{N}$ let $\langle \alpha_n, \beta_n \rangle := \omega^*(\langle \varepsilon, z \rangle, \langle u, v \rangle^n)$, let $X = \{\langle \alpha_n, \beta_n \rangle \mid n \in \mathbb{N}\}$. Assume that all advances in X are balancible (cf. Definition 4.6.1). Then*

1. *X is infinite iff $|u| \neq |v|$,*
2. *if $|u| \neq |v|$, then there exists a number $n_0 \in \mathbb{N}$ such that for any $n > n_0$ we have $|\alpha_n| + |\beta_n| \geq (n - n_0)|(|u| - |v|)|$,*
3. *X is finite iff there exist $t \in \Sigma^*$ and $k \in \mathbb{N}$ such that $|t| < |u|$ and $z = u^k t$ and $ut = tv$,*

4. if X is finite, then $|X| = 1$.

Proof. (Claims 1., 2.) Assume that $|u| = |v|$. Since $\langle \alpha_{n+1}, \beta_{n+1} \rangle$ are balancible and $\langle \alpha_{n+1}, \beta_{n+1} \rangle = \omega(\langle \alpha_n, \beta_n \rangle, \langle u, v \rangle)$ we have

$$|\alpha_{n+1}| + |\beta_{n+1}| = \begin{cases} |(\alpha_n \cdot u)^{-1}(\beta_n \cdot v)| & \text{if } \alpha_n \cdot u \text{ is a prefix of } \beta_n \cdot v, \\ |(\beta_n \cdot v)^{-1}(\alpha_n \cdot u)| & \text{if } \beta_n \cdot v \text{ is a prefix of } \alpha_n \cdot u. \end{cases}$$

Therefore $|\alpha_{n+1}| + |\beta_{n+1}| = |(|\alpha_n \cdot u| - |\beta_n \cdot v|)| = |(|\alpha_n| + |u| - |\beta_n| - |v|)| = |(|\alpha_n| - |\beta_n|)|$. Since $|\alpha_n| = 0$ or $|\beta_n| = 0$ we have $|\alpha_n| + |\beta_n| = |z|$ for all $n \in \mathbb{N}$. Since the alphabet Σ is finite and all elements of X are balancible there are at most $2|\Sigma|^{|z|}$ values in X and therefore X is finite.
Let $|u| \neq |v|$. We have two cases.
Case 1: $|v| > |u|$. In this case $\langle \alpha_1, \beta_1 \rangle = \omega(\langle \varepsilon, z \rangle, \langle u, v \rangle) = \langle \varepsilon, u^{-1}zv \rangle$, because zv cannot be a prefix of u. We obtain $|\alpha_1| + |\beta_1| = |u^{-1}zv| = |z| + |v| - |u|$. Similarly $\langle \alpha_2, \beta_2 \rangle = \omega(\langle \varepsilon, u^{-1}zv \rangle, \langle u, v \rangle) = \langle \varepsilon, u^{-2}zv^2 \rangle$ and $|\alpha_2| + |\beta_2| = |u^{-2}zv^2| = |z| + 2(|v| - |u|)$. Iterating, for any $n \in \mathbb{N}$ we obtain $\langle \alpha_n, \beta_n \rangle = \langle \varepsilon, u^{-n}zv^n \rangle$ and

$$|\alpha_n| + |\beta_n| = |u^{-n}zv^n| = |z| + n(|v| - |u|).$$

Therefore X is infinite and for $n_0 := 0$ Claim 2 above is fulfilled.
Case 2: $|v| < |u|$. Clearly $\alpha_0 = \varepsilon$. Consider the values $n = 0, 1, 2 \ldots$. As long as $\alpha_n = \alpha_{n+1} = \varepsilon$ it follows from $\langle \alpha_{n+1}, \beta_{n+1} \rangle = \omega(\langle \alpha_n, \beta_n \rangle, \langle u, v \rangle)$ that $\beta_{n+1} = u^{-1}\beta_n v$. Therefore $|\beta_{n+1}| = |\beta_n| - (|u| - |v|)$ and $|\beta_{n+1}| < |\beta_n|$. Hence there exists $n_0 \in \mathbb{N}$ such that $\beta_{n_0} = \varepsilon$. Then $\langle \alpha_{n_0+1}, \beta_{n_0+1} \rangle = \omega(\langle \alpha_{n_0}, \varepsilon \rangle, \langle u, v \rangle) = \langle v^{-1}\alpha_{n_0}u, \varepsilon \rangle$ because $\alpha_{n_0}u$ cannot be a prefix of v. Clearly $|\alpha_{n_0+1}| + |\beta_{n_0+1}| = |\alpha_{n_0}| + |u| - |v|$. For any $m \in \mathbb{N}^+$ we obtain $\langle \alpha_{n_0+m}, \beta_{n_0+m} \rangle = \langle v^{-m}\alpha_{n_0}u^m \rangle$ and

$$|\alpha_{n_0+m}| + |\beta_{n_0+m}| = |\alpha_{n_0}| + m(|u| - |v|).$$

Therefore X is infinite and Claim 2 is satisfied.
(Claim 3, "$\Leftarrow$") Let $z = u^k t$, and $ut = tv$. Then u is a prefix of $zv = u^k tv = u^k ut$ and

$$\omega(\langle \varepsilon, z \rangle, \langle u, v \rangle) = \langle \varepsilon, u^k t \rangle = \langle \varepsilon, z \rangle . \qquad (\$)$$

Clearly in this case $|X| = 1$.
(Claim 3, "$\Rightarrow$") Let X be finite. It follows (see 1.) that $|u| = |v|$. If $z = \varepsilon$, then $u = v$, thus $t := \varepsilon$, $k := 0$ satisfy the properties described in Claim 3. Let us now assume that $z \neq \varepsilon$. Since $uv \neq \varepsilon$ and $|u| = |v|$ we have $u \neq \varepsilon$. In this case $\langle \alpha_1, \beta_1 \rangle = \langle \varepsilon, u^{-1}zv \rangle$, because zv cannot be a prefix of u. Clearly $|u^{-1}zv| = |z|$. At the next step we have $\langle \alpha_2, \beta_2 \rangle = \langle \varepsilon, u^{-2}zv^2 \rangle$ and $|\beta_2| = |z|$. After n steps we have $\langle \alpha_n, \beta_n \rangle = \langle \varepsilon, u^{-n}zv^n \rangle$ and $|\beta_n| = |z|$. Since X is finite there exist $h < l \in \mathbb{N}$ such that

$$\begin{aligned} u^{-h}zv^h &= u^{-l}zv^l \\ u^{l-h}zv^h &= zv^l. \end{aligned}$$

Since both sides of the equation have a common suffix v^h, with $m := l - h > 0$ we get

$$u^m z = zv^m.$$

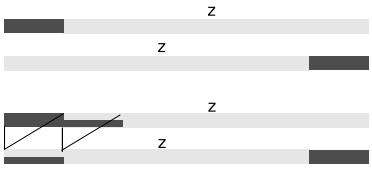

Figure 5.4: Solutions of an equation $u^m z = z v^m$ where $z, u \neq \varepsilon$, $m > 0$ and $|u^m| < |z|$. The situation is shown in the upper part and leads to two overlapping copies of z. The lower copy of z starts with u^m. Since both copies of z are identical, the upper string (left-hand side of equation) starts with $u^m u^m$. Continuing with this copy-paste mechanism shows that z has the form $u^k t$ where t is a possibly empty proper prefix of u.

From the last equation it follows that z can be represented in the form $z = u^k t$ where t is a proper prefix of u. This is obvious if $|u^m| \geq |z|$. For the situation $|u^m| < |z|$ see Figure 5.4. From $u^m z = z v^m$ we obtain

$$\begin{aligned} u^{m+k} t &= u^k t v^m \\ u^m t &= t v^m. \end{aligned}$$

Since t is a proper prefix of u, looking at the two prefixes of length $|t| + |u| = |t| + |v|$ of the last equation we obtain $ut = tv$.
Claim 4 of the lemma follows directly from Claim 3, cf. (\$). □

Lemma 5.3.2 *Let $x, y, \alpha, \beta \in \Sigma^*$. If $\omega(\langle x, y\rangle, \langle \alpha, \beta\rangle) = \langle u, v\rangle$ and $x \wedge y = \varepsilon$, then $|u| + |v| \geq |x| + |y| - (|\alpha| + |\beta|)$.*

Proof. Case 1: Let $x \neq \varepsilon$ and $y \neq \varepsilon$. Then $c = x\alpha \wedge y\beta = \varepsilon$ and therefore

$$|u| + |v| = |c^{-1} x\alpha| + |c^{-1} y\beta| = |x| + |y| + |\alpha| + |\beta| \geq |x| + |y| - (|\alpha| + |\beta|).$$

Case 2: Let $x = \varepsilon$. Then $|c| = |\alpha \wedge y\beta| \leq |\alpha|$ and therefore

$$|u| + |v| = |c^{-1} x\alpha| + |c^{-1} y\beta| \geq |x| + |y| + |\alpha| + |\beta| - 2|\alpha| \geq |x| + |y| - (|\alpha| + |\beta|).$$

Case 3: Let $y = \varepsilon$. Then $|c| = |x\alpha \wedge \beta| \leq |\beta|$ and therefore

$$|u| + |v| = |c^{-1} x\alpha| + |c^{-1} y\beta| \geq |x| + |y| + |\alpha| + |\beta| - 2|\beta| \geq |x| + |y| - (|\alpha| + |\beta|).$$ □

Recall that $Adm(q)$ denotes the set of all admissible advances of a state q of a squared output transducer (cf. Definition 4.6.6).

Lemma 5.3.3 *Let $\mathcal{T}$ be a classical real-time finite-state transducer, let q denote a state of the squared output transducer $\mathcal{S}_\mathcal{T}$ of $\mathcal{T}$.*

1. *If $\langle u, v\rangle \in Adm(q)$ and the corresponding path in $\mathcal{S}_\mathcal{T}$ is $\pi = q_0 \to \ldots \overset{\langle \alpha, \beta\rangle}{\to} q$, then $d_S(\alpha, \beta) = |u| + |v|$.*
2. *If $Adm(q)$ is finite and there exists $\langle u, v\rangle \in Adm(q)$ such that $\langle u, v\rangle$ is not balancible, then each loop $\langle q, \langle \alpha, \beta\rangle, q\rangle \in \Delta'^*$ has label $\langle \alpha, \beta\rangle = \langle \varepsilon, \varepsilon\rangle$.*

Proof. (1.) This statement follows from the fact that $\langle u, v\rangle = \omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha, \beta\rangle)$ and the definitions of the advance function and d_S.
(2.) Since $\langle u, v\rangle$ is not balancible we have $\omega^*(\langle u, v\rangle, \langle \alpha, \beta\rangle^n) = \langle u\alpha^n, v\beta^n\rangle$ for all $n \in \mathbb{N}$. Hence, if $\langle \alpha, \beta\rangle \neq \langle \varepsilon, \varepsilon\rangle$, then the set $Adm(q)$ is infinite. □

Theorem 5.3.4 *Let $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta\rangle$ be a trimmed functional classical real-time finite-state transducer and let $\mathcal{S}_\mathcal{T}$ be the squared output transducer of $\mathcal{T}$. Then $\mathcal{T}$ has the bounded variation property iff for each state q of $\mathcal{S}_\mathcal{T}$ the set of admissible advances $Adm(q)$ is finite.*

Proof. ("⇐") Let $Adm(q)$ be finite for each state $q \in Q \times Q$. We define

$$L := \max_{q \in Q\times Q, \langle \alpha,\beta\rangle \in Adm(q)} |\alpha| + |\beta|$$

as the length of the longest advance. Let

$$C := \max_{\langle q', \langle \sigma,\alpha\rangle, q''\rangle \in \Delta} |\alpha|$$

denote the length of the longest transition output in $\mathcal{T}$. To prove the bounded variation property let $u, v \in dom(L_\mathcal{T})$, $z = u \wedge v$, $u = zu'$, $v = zv'$ and $d_S(u, v) = |u'| + |v'|$. Consider any pair of successful paths in $\mathcal{T}$:

$$\begin{aligned} \pi_1 &= i \to \ldots^{\langle z,\alpha\rangle} \to p \to \ldots^{\langle u',\alpha'\rangle} \to t \\ \pi_2 &= j \to \ldots^{\langle z,\beta\rangle} \to q \to \ldots^{\langle v',\beta'\rangle} \to s. \end{aligned}$$

There exists a path in $\mathcal{S}_\mathcal{T}$:

$$\pi = \langle i, j\rangle \to \ldots^{\langle \alpha,\beta\rangle} \to \langle p, q\rangle$$

and a corresponding admissible advance of $\langle p, q\rangle$

$$\omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha, \beta\rangle) = \langle h_1, h_2\rangle.$$

Case 1: The advance $h = \langle h_1, h_2\rangle$ is balancible.
Proposition 4.6.3 Point 3 shows that $\alpha h_2 = \beta h_1$. Since one of the strings h_1, h_2 is empty and we have $d_S(\alpha, \beta) \leq |h_1| + |h_2|$ we obtain

$$\begin{aligned} d_S(L_\mathcal{T}(u), L_\mathcal{T}(v)) &= d_S(\alpha\alpha', \beta\beta') \\ &\leq d_S(\alpha, \beta) + |\alpha'| + |\beta'| \\ &\leq |h_1| + |h_2| + |\alpha'| + |\beta'| \\ &\leq L + C(|u'| + |v'|) \\ &= L + C \cdot d_S(u, v). \end{aligned}$$

Case 2: The advance $\langle h_1, h_2\rangle$ is not balancible. Since $\langle h_1, h_2\rangle$ is not balancible z can be represented as $z = z_1 a z_2$, where $a \in \Sigma_I$, $z_1, z_2 \in \Sigma_I^*$ such that the paths are factorized in the following way:

$$\begin{aligned} \pi_1 &= i \to \ldots^{\langle z_1,\alpha_1\rangle} \to p_1 \to^{\langle a,x\rangle} p_2 \to \ldots^{\langle z_2,\alpha_2\rangle} \to p \to \ldots^{\langle u',\alpha'\rangle} \to t \\ \pi_2 &= j \to \ldots^{\langle z_1,\beta_1\rangle} \to q_1 \to^{\langle a,y\rangle} q_2 \to \ldots^{\langle z_2,\beta_2\rangle} \to q \to \ldots^{\langle v',\beta'\rangle} \to s \end{aligned}$$

and

$$\pi = \langle i,j\rangle \to \ldots^{\langle \alpha_1,\beta_1\rangle} \to \langle p_1,q_1\rangle \to^{\langle x,y\rangle} \langle p_2,q_2\rangle \to \ldots^{\langle \alpha_2,\beta_2\rangle} \to \langle p,q\rangle$$

such that $\langle h_1', h_2'\rangle = \omega(\langle\varepsilon,\varepsilon\rangle, \langle\alpha_1,\beta_1\rangle)$ is balancible and $\omega(\langle\varepsilon,\varepsilon\rangle, \langle\alpha_1 x, \beta_1 y\rangle)$ is not balancible. Each state on the path

$$\pi' = \langle p_2,q_2\rangle \to \ldots^{\langle \alpha_2,\beta_2\rangle} \langle p,q\rangle$$

has an admissible advance that is not balancible. From Lemma 5.3.3, Point 2 it follows that if a transition of π' is part of a loop, then the label of the loop is $\langle\varepsilon,\varepsilon\rangle$. The number of transitions on π' that do not belong to a loop is bounded by $|Q|^2$. Therefore

$$\begin{aligned} d_S(L_{\mathcal{T}}(u), L_{\mathcal{T}}(v)) &= d_S(\alpha_1 x \alpha_2 \alpha', \beta_1 y \beta_2 \beta') \\ &\le |h_1| + |h_2| + |x| + |y| + |\alpha_2| + |\beta_2| + |\alpha'| + |\beta'| \\ &\le L + 2C(|Q|^2+1) + C(|u'| + |v'|) \\ &= L + 2C(|Q|^2+1) + C \cdot d_S(u,v). \end{aligned}$$

("⇒") Let $\mathcal{T}$ have the bounded variation property. Assume there exists a state $\langle p,q\rangle \in Q \times Q$ such that $Adm(\langle p,q\rangle)$ is infinite. Since $\mathcal{T}$ is trimmed there exist two paths

$$\begin{aligned} \pi_1' &= p \to \ldots^{\langle u',\alpha'\rangle} \to t, \\ \pi_2' &= q \to \ldots^{\langle v',\beta'\rangle} \to s \end{aligned}$$

such that $t,s \in F$. Let $k = |u'| + |v'|$. For each admissible advance $\langle x,y\rangle \in Adm(\langle p,q\rangle)$ there exist two paths

$$\begin{aligned} \pi_1 &= i \to \ldots^{\langle z,\alpha\rangle} \to p \\ \pi_2 &= j \to \ldots^{\langle z,\beta\rangle} \to q \end{aligned}$$

such that $\langle x,y\rangle = \omega(\langle\varepsilon,\varepsilon\rangle, \langle\alpha,\beta\rangle)$. Therefore $O_{\mathcal{T}}(zu') = \alpha\alpha'$ and $O_{\mathcal{T}}(zv') = \beta\beta'$. We have $d_S(zu', zv') \le |u'| + |v'| = k$. On the other hand by Lemma 5.3.3 Point 1 and Lemma 5.3.2 we obtain $d_S(\alpha\alpha', \beta\beta') = \omega(\langle x,y\rangle, \langle\alpha',\beta'\rangle) \ge |x| + |y| - (|\alpha'| + |\beta'|)$. The set $Adm(\langle p,q\rangle)$ is infinite and therefore the sum $|x|+|y|$ for the admissible advance $\langle x,y\rangle \in Adm(\langle p,q\rangle)$ can be arbitrarily large. The words α' and β' are fixed and therefore the bounded variation property is violated. Hence the set of admissible advances is finite. □

The following lemma provides us with an effective way to check whether the set of admissible advances $Adm(q)$ of a state $q \in Q \times Q$ is finite.

Lemma 5.3.5 *Let $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta\rangle$ be a trimmed functional classical real-time finite-state transducer, let $C := \max_{\langle q',\langle\sigma,\alpha\rangle,q''\rangle\in\Delta} |\alpha|$. Let $\langle u,v\rangle$ be an admissible advance of the state $\langle p,q\rangle \in Q \times Q$ of the squared output transducer $\mathcal{S}_{\mathcal{T}}$. If $\mathcal{T}$ has the bounded variation property, then $|u| < C|Q|^2$ and $|v| < C|Q|^2$.*

Proof. Consider a shortest path π in $\mathcal{S}_{\mathcal{T}}$ from an initial state $\langle i,j\rangle \in I \times I$ to $\langle p,q\rangle$ with a label $\langle\alpha,\beta\rangle$ such that $\omega(\langle\varepsilon,\varepsilon\rangle, \langle\alpha,\beta\rangle) = \langle u,v\rangle$. If we assume that

the path is longer than $|Q|^2$, then we find a pair of states $\langle r, s\rangle$ twice on the path and π has the form

$$\begin{aligned}\pi &= \langle i, j\rangle \to \ldots^{\langle \alpha_1, \beta_1\rangle} \\ &\to \langle r, s\rangle \to \ldots^{\langle \alpha_2, \beta_2\rangle} \to \langle r, s\rangle \\ &\to \ldots^{\langle \alpha_3, \beta_3\rangle} \to \langle p, q\rangle\,.\end{aligned}$$

Let $\langle u_1, v_1\rangle = \omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha_1, \beta_1\rangle)$. Consider the subpath

$$\langle r, s\rangle \to \ldots^{\langle \alpha_2, \beta_2\rangle} \to \langle r, s\rangle\,.$$

If $\alpha_2 = \beta_2 = \varepsilon$, then $\omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha_1\alpha_2, \beta_1\beta_2\rangle) = \langle u_1, v_1\rangle$ and therefore π is not a shortest path to $\langle p, q\rangle$ such that $\omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha, \beta\rangle) = \langle u, v\rangle$. Hence $\alpha_2\beta_2 \neq \varepsilon$. If after k repetitions of the above subpath

$$\langle r, s\rangle \to \ldots^{\langle \alpha_2, \beta_2\rangle} \to \langle r, s\rangle \to \ldots^{\langle \alpha_2, \beta_2\rangle} \ldots\ldots\ldots^{\langle \alpha_2, \beta_2\rangle} \to \langle r, s\rangle$$

the advance $\omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha_1\alpha_2^k, \beta_1\beta_2^k\rangle)$ is not balancible, then $Adm(\langle r, s\rangle)$ is infinite (cf. Lemma 5.3.3 Point 2), which contradicts the bounded variation property (cf. Theorem 5.3.4). Hence $X = \{\omega^*(\langle u_1, v_1\rangle, \langle \alpha_2, \beta_2\rangle^k) \mid k \in \mathbb{N}\}$ only contains balancible advances. Lemma 5.3.1 shows that in this case $\omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha_1\alpha_2, \beta_1\beta_2\rangle) = \omega(\langle u_1, v_1\rangle, \langle \alpha_2, \beta_2\rangle) = \langle u_1, v_1\rangle$. This is a contradiction since π is a shortest path. Therefore the length of π is smaller than $|Q|^2$. Since $\langle u, v\rangle = \omega(\langle \varepsilon, \varepsilon\rangle, \langle \alpha, \beta\rangle)$ we obtain $|u| \leq |\alpha| < C|Q|^2$ and $|v| \leq |\beta| < C|Q|^2$. □

Procedure for deciding the bounded variation property. If the trimmed functional classical finite-state transducer $\mathcal{T}$ is given we start with building the corresponding squared output transducer with all reachable states first and constructing the admissible advances (cf. Corollary 4.6.11) for its states afterwards. If during the construction an admissible advance $\langle u, v\rangle$ is generated such that $|u| \geq C|Q|^2$ or $|v| \geq C|Q|^2$ we stop. In this case $\mathcal{T}$ does not have the bounded variation property (Lemma 5.3.5). In the other case the procedure will stop since the alphabet is finite and hence the number of possible admissible advances that can be generated is bounded. In this case according to Theorem 5.3.4 the transducer $\mathcal{T}$ has the bounded variation property.

The following observations show how to include the bounded variation test directly in the determinization procedure.

Proposition 5.3.6 *Let $U = \{u_1, u_2, \ldots, u_n\} \subset \Sigma^*$ be a non-empty set of words and $u = \bigwedge_{k=1}^{n} u_k$. Then for every index $i \in \{1, \ldots, n\}$ there exists a index $j \in \{1, \ldots, n\}$ such that $u = u_i \wedge u_j$.*

Proof. Consider the word u_i. If $u_i = u$, then u_i is a prefix of each u_j, for any j we have $u = u_i \wedge u_j$. Otherwise u_i has the form $u\sigma v$ for some $\sigma \in \Sigma$ and $v \in \Sigma^*$. If there exists $j \in \{1, \ldots, n\}$ such that $u_j = u$ we have $u = u_i \wedge u_j$. Otherwise all u_j are longer than u. If all u_j would have a prefix $u\sigma$, then the longest common prefix of all words would start with $u\sigma$, a contradiction. Hence there exists $j \in \{1, \ldots, n\}$ with a prefix $u\sigma'$ where $\sigma \neq \sigma'$ and we get $u = u_i \wedge u_j$. □

Corollary 5.3.7 *Let $\mathcal{T} = \langle \Sigma_I^* \times \Sigma^*, Q, I, F, \Delta\rangle$ be a trimmed functional classical real-time finite-state transducer, let $\mathcal{S}_\mathcal{T} = \langle \Sigma^* \times \Sigma^*, Q \times Q, I \times I, F \times F, \Delta'\rangle$*

be the squared output transducer of $\mathcal{T}$. *Let* $\mathcal{T}'^{(n)} = \langle \Sigma_I, \Sigma^*, Q', q'_0, F', \delta', \lambda', \Psi' \rangle$ *be constructed in* n *steps by the inductive determinization procedure described in Section 5.2. Let* $S \in Q'$ *be a state in* $\mathcal{T}'^{(n)}$. *Then for each pair* $\langle p, u \rangle \in S$ *there exists a pair* $\langle q, v \rangle \in S$ *such that* $\langle u, v \rangle$ *is an admissible advance of the state* $\langle p, q \rangle$ *(i.e* $\langle u, v \rangle \in Adm(\langle p, q \rangle)$*).*

Proof. Since S is a state in $\mathcal{T}'^{(n)}$ it follows from Lemma 5.2.2 that there exists a word $w \in \Sigma_I^*$ such that

$$S = \delta^*(q'_0, w) = \{\langle q, \gamma \rangle \mid \exists q_0 \in I, \langle q_0, \langle w, \alpha \rangle, q \rangle \in \Delta^* : \gamma = z^{-1}\alpha\},$$

where $z = \bigwedge_{q_0 \in I, \langle q_0, \langle w, \alpha \rangle, q \rangle \in \Delta^*} \alpha$ is the longest common prefix for the outputs produce with input w. Since $\langle p, u \rangle \in S$ there exists a state $q_{01} \in I$ and a generalized transition $\langle q_{01}, \langle w, \alpha \rangle, p \rangle \in \Delta^*$ such that $u = z^{-1}\alpha$. From Proposition 5.3.6 it follows that there exists a state $q_{02} \in I$ and a generalized transition $\langle q_{02}, \langle w, \beta \rangle, q \rangle \in \Delta^*$ such that $z = \alpha \wedge \beta$. We have $\langle q, v \rangle \in S$, where $v = z^{-1}\beta$. But then $\langle \langle q_{01}, q_{02} \rangle, \langle \alpha, \beta \rangle, \langle p, q \rangle \rangle \in \Delta'^*$ and thus $\langle u, v \rangle \in Adm(\langle p, q \rangle)$. □

The following theorem follows from Corollary 5.3.7, Lemma 5.3.5, Theorem 5.3.4 and Theorem 5.2.6.

Theorem 5.3.8 *Let* $\mathcal{T} = \langle \Sigma^* \times \Sigma^*, Q, I, F, \Delta \rangle$ *be a trimmed real-time functional classical transducer, let* $C := \max_{\langle q', \langle \sigma, \alpha \rangle, q'' \rangle \in \Delta} |\alpha|$. *Then the inductive construction presented in Section 5.2 terminates iff at each step* n *of the construction of the corresponding subsequential transducer*

$$\mathcal{T}'^{(n)} = \langle \Sigma_I, \Sigma^*, Q', q'_0, F', \delta', \lambda', \Psi' \rangle$$

for each state S *in* Q' *and each pair* $\langle p, u \rangle \in S$ *we have* $|u| < C|Q|^2$.

Remark 5.3.9 The above theorem provides us with an effective way for testing the termination of the inductive determinization construction presented in Section 5.2. For ensuring the termination it is sufficient to test during the construction whether the lengths of the delays in the states are within the limit.

5.4 Minimal subsequential finite-state transducers - Myhill-Nerode relation for subsequential transducers

When we studied minimization of deterministic automata we found that the Myhill-Nerode relation (Definition 3.4.8) gives a direct (mathematical) way for defining a minimal deterministic automaton. Since the language of a deterministic automaton is a set of strings, the Myhill-Nerode relation was introduced as an equivalence relation between strings, determined by a fixed string language. We now show that also in the case of subsequential finite-state transducers a new kind of Myhill-Nerode relation yields a minimal subsequential transducer. Since subsequential transducers represent string functions, the Myhill-Nerode relation is now determined by a fixed partial string function f. Roughly, two

strings u and v are defined to be equivalent with respect to f if all common extensions $u \cdot w$ and $v \cdot w$ are "treated in the same way" by f: either f is undefined for both extensions or $f(u \cdot w)$ and $f(v \cdot w)$ have identical suffixes. However, this basic idea needs to be refined. We follow the approach in [Mohri, 2000].

Definition 5.4.1 Let $f : \Sigma^* \to \Sigma'^*$ be a (partial) function. Then

$$\begin{array}{lll} R_f = \{\langle u, v\rangle \in \Sigma^* \times \Sigma^* \mid & \exists u' \in \Sigma'^* \ \exists v' \in \Sigma'^* & \forall w \in \Sigma^* : \\ & (u \cdot w \in dom(f) \leftrightarrow & v \cdot w \in dom(f)) \ \& \\ & (u \cdot w \in dom(f) \rightarrow & u' \in Pref(f(u \cdot w)) \ \& \ v' \in Pref(f(v \cdot w)) \ \& \\ & & u'^{-1} f(u \cdot w) = v'^{-1} f(v \cdot w))\} \end{array}$$

is called the *Myhill-Nerode relation* for f.

Note that R_f acts on the full set $\Sigma^* \times \Sigma^*$ despite of the fact that f can be partial.

Proposition 5.4.2 *Let $f : \Sigma^* \to \Sigma'^*$ be a function. Then the Myhill-Nerode relation for f is a right invariant equivalence relation.*

Proof. Clearly R_f is reflexive and symmetric. We show that R_f is transitive. Let $u_1 \ R_f \ u_2$ and $u_2 \ R_f \ u_3$ for strings $u_1, u_2, u_3 \in \Sigma^*$. Then

$$u_1 \cdot w \in dom(f) \leftrightarrow u_2 \cdot w \in dom(f) \leftrightarrow u_3 \cdot w \in dom(f)$$

for all $w \in \Sigma^*$ and there exist strings $u_1', u_2', u_2'', u_3' \in \Sigma^*$ such that for all $w \in \Sigma^*$ we have

$$\begin{array}{l} u_1 \cdot w \in dom(f) \rightarrow u_1'^{-1} f(u_1 \cdot w) = u_2'^{-1} f(u_2 \cdot w) \\ u_2 \cdot w \in dom(f) \rightarrow u_2''^{-1} f(u_2 \cdot w) = u_3'^{-1} f(u_3 \cdot w). \end{array}$$

We have to show that there exist strings v_1, v_3 such that for all $w \in \Sigma^*$ such that $u_1 \cdot w \in dom(f)$ we have $v_1^{-1} f(u_1 \cdot w) = v_3^{-1} f(u_3 \cdot w)$. We may assume that there exists $w_0 \in \Sigma^*$ such that $u_1 \cdot w_0 \in dom(f)$. Then

$$\begin{array}{rcl} u_1'^{-1} f(u_1 \cdot w_0) & = & u_2'^{-1} f(u_2 \cdot w_0) \\ u_2''^{-1} f(u_2 \cdot w_0) & = & u_3'^{-1} f(u_3 \cdot w_0). \end{array}$$

These equations imply that both u_2' and u_2'' are prefixes of $f(u_2 \cdot w_0)$. We assume that $|u_2'| \geq |u_2''|$ (the other case is similar). Let $u_2' = u_2'' u$. Since $u_2'' u$ is a prefix of $f(u_2 \cdot w_0)$ the second equation implies that $f(u_3 \cdot w_0)$ has a prefix $u_3' u$. Let $v_1 := u_1', v_3 := u_3' u$. Let $w \in \Sigma^*$ be any word such that $u_1 \cdot w \in dom(f)$ and $u_3 \cdot w \in dom(f)$. Then

$$\begin{array}{rcl} u_1'^{-1} f(u_1 \cdot w) & = & u_2'^{-1} f(u_2 \cdot w) \\ u_2''^{-1} f(u_2 \cdot w) & = & u_3'^{-1} f(u_3 \cdot w). \end{array}$$

Since $u_2' = u_2'' u$ is a prefix of $f(u_2 \cdot w)$ it follows that $u_3' u$ is a prefix of $f(u_3 \cdot w)$ and

$$u_1'^{-1} f(u_1 \cdot w) = u_2'^{-1} f(u_2 \cdot w) = (u_2'' u)^{-1} f(u_2 \cdot w) = (u_3' u)^{-1} f(u_3 \cdot w).$$

This shows that R_f is transitive and an equivalence relation. It remains to show that R_f is right invariant. Let $u\ R_f\ v$ and $z \in \Sigma^*$. In order to prove that $u{\cdot}z\ R_f\ v{\cdot}z$ we have to show that there exist $u', v' \in \Sigma'^*$ such that for any $w \in \Sigma^*$ we have (a) $(u{\cdot}z){\cdot}w \in dom(f)$ iff $(v{\cdot}z){\cdot}w \in dom(f)$, and (b) if $(u{\cdot}z){\cdot}w \in dom(f)$, then $u'^{-1}f((u \cdot z) \cdot w) = v'^{-1}f((v \cdot z) \cdot w)$. Let $w \in \Sigma^*$. Since $u\ R_f\ v$ there exist $u', v' \in \Sigma'^*$ such that for $w' = z \cdot w$ we have (a) $u \cdot w' = (u \cdot z) \cdot w \in dom(f)$ iff $v \cdot w' = (v \cdot z) \cdot w \in dom(f)$, and (b) if $u \cdot w' = (u \cdot z) \cdot w \in dom(f)$, then $u'^{-1}f(u \cdot w') = u'^{-1}f((u \cdot z) \cdot w) = v'^{-1}f(v \cdot w') = v'^{-1}f((v \cdot z) \cdot w)$. □

Definition 5.4.3 Let $f : \Sigma^* \to \Sigma'^*$ be a function. The *longest common output* of f is the function $lco_f : \Sigma^* \to \Sigma'^*$ defined as follows:

$$lco_f(u) = \begin{cases} \bigwedge_{w \in \Sigma^* \ \&\ u \cdot w \in dom(f)} f(u \cdot w) & \text{if } u \in Pref(dom(f)) \\ \varepsilon & \text{otherwise.} \end{cases}$$

In order to simplify the construction of the Myhill-Nerode transducer for a subsequential function we introduce a non-significant extension of the subsequential finite-state transducer by adding an output in front of all outputs.

Definition 5.4.4 A *subsequential finite-state transducer with initial output* over the monoid $\mathcal{M} = \langle M, \circ, e\rangle$ is a tuple $\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \iota, \Psi\rangle$ such that $\langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \Psi\rangle$ is a subsequential finite-state transducer and $\iota \in M$. The output function of the subsequential finite-state transducer with initial output $\mathcal{T}$ is defined as $O_{\mathcal{T}}(\alpha) = \iota \circ \lambda^*(q_0, \alpha) \circ \Psi(\delta^*(q_0, \alpha))$.

As before, two subsequential finite-state transducers with initial output are called *equivalent* if the output functions of the transducers coincide. Clearly, any subsequential finite-state transducer can be considered as a subsequential finite-state transducer with initial output $\iota = \varepsilon$. The following proposition shows that by adding a new initial state every subsequential transducer with initial output can be converted to an ordinary subsequential transducer.

Proposition 5.4.5 *Let $\mathcal{T} = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda, \iota, \Psi\rangle$ be a subsequential finite-state transducer with initial output. Let $\mathcal{T}' = \langle \Sigma, \mathcal{M}, Q \cup \{q_0'\}, q_0', F', \delta', \lambda', \Psi'\rangle$ be the subsequential transducer where*

- *q_0' is a new starting state,*
- *$\delta' = \delta \cup \{\langle q_0', \sigma, q\rangle \mid \langle q_0, \sigma, q\rangle \in \delta\}$,*
- *$\lambda' = \lambda \cup \{\langle q_0', \sigma, \iota \circ \alpha\rangle \mid \langle q_0, \sigma, \alpha\rangle \in \lambda\}$,*
- *$F' = F \cup \begin{cases} \{q_0'\} & \text{if } q_0 \in F \\ \emptyset & \text{otherwise,} \end{cases}$*
- *$\Psi' = \Psi \cup \begin{cases} \{\langle q_0', \iota \circ \Psi(q_0)\rangle\} & \text{if } q_0 \in F \\ \emptyset & \text{otherwise.} \end{cases}$*

Then we have $O_{\mathcal{T}} = O_{\mathcal{T}'}$.

For the next proposition we assume that the transition function of a transducer is total.

Proposition 5.4.6 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a classical subsequential finite-state transducer with initial output representing $f : \Sigma^* \to \Sigma'^*$ and total transition function δ. Let*

$$R_{\mathcal{T}} := \{\langle u, v \rangle \in \Sigma^* \times \Sigma^* \mid \delta^*(q_0, u) = \delta^*(q_0, v)\}.$$

Then $R_{\mathcal{T}}$ is a right invariant equivalence relation and $R_{\mathcal{T}}$ is a refinement of the Myhill-Nerode relation R_f.

Proof. The right invariance of $R_{\mathcal{T}}$ is proven similarly as in Proposition 3.4.7. Let $u \; R_{\mathcal{T}} \; v$. Let $u' = \iota \cdot \lambda^*(q_0, u)$, $v' = \iota \cdot \lambda^*(q_0, v)$ and $w \in \Sigma^*$ be an arbitrary word. Then

$$\delta^*(q_0, u \cdot w) = \delta^*(\delta^*(q_0, u), w) = \delta^*(\delta^*(q_0, v), w) = \delta^*(q_0, v \cdot w).$$

Hence

$$\begin{aligned} u \cdot w \in dom(f) &\Leftrightarrow \delta^*(q_0, u \cdot w) \in F \\ &\Leftrightarrow \delta^*(q_0, v \cdot w) \in F \\ &\Leftrightarrow v \cdot w \in dom(f). \end{aligned}$$

Moreover, if $u \cdot w \in dom(f)$, then $p := \delta^*(q_0, u \cdot w) = \delta^*(q_0, v \cdot w)$ is final and

$$f(u \cdot w) = \iota \cdot \lambda^*(q_0, u \cdot w) \cdot \Psi(p) = \iota \cdot \lambda^*(q_0, u) \cdot \lambda^*(\delta^*(q_0, u), w) \cdot \Psi(p)$$

and

$$f(v \cdot w) = \iota \cdot \lambda^*(q_0, v \cdot w) \cdot \Psi(p) = \iota \cdot \lambda^*(q_0, v) \cdot \lambda^*(\delta^*(q_0, v), w) \cdot \Psi(p)$$

and therefore

$$u'^{-1} f(u \cdot w) = \lambda^*(\delta^*(q_0, u), w) \cdot \Psi(p) = \lambda^*(\delta^*(q_0, v), w) \cdot \Psi(p) = v'^{-1} f(v \cdot w).$$

□

Corollary 5.4.7 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a subsequential classical finite-state transducer with initial output and total transition function δ representing $f : \Sigma^* \to \Sigma'^*$. Then $|\Sigma^*/R_f| \leq |\Sigma^*/R_{\mathcal{T}}| = |Q|$.*

Proposition 5.4.8 *Let $f : \Sigma^* \to \Sigma'^*$ be a function such that the index of the Myhill-Nerode relation R_f is finite. Let*

$$\mathcal{T}_f = \left\langle \Sigma, \Sigma', \Sigma^*/R_f, [\varepsilon]_{R_f}, \{[s]_{R_f} \mid s \in dom(f)\}, \delta, \lambda, \iota, \Psi \right\rangle,$$

where:

- $\delta = \{\left\langle [u]_{R_f}, a, [u \cdot a]_{R_f} \right\rangle \mid u \in \Sigma^*, a \in \Sigma\}$,
- $\lambda = \{\left\langle [u]_{R_f}, a, lco_f(u)^{-1} lco_f(u \cdot a) \right\rangle \mid u \in \Sigma^*, a \in \Sigma, u \cdot a \in Pref(dom(f))\} \cup$ $\{\left\langle [u]_{R_f}, a, \varepsilon \right\rangle \mid u \in \Sigma^*, a \in \Sigma, u \cdot a \notin Pref(dom(f))\}$,
- $\iota = lco_f(\varepsilon)$,

- $\Psi = \{\langle [u]_{R_f}, lco_f(u)^{-1} f(u)\rangle \mid u \in dom(f)\}$.

Then $\mathcal{T}_f$ is a classical subsequential finite-state transducer with initial output and we have $O_{\mathcal{T}_f} = f$ and the transition function δ is total.

Proof. We have to show that the transducer $\mathcal{T}_f$ is well-defined. Clearly, Q and F are finite since R_f has a finite index.

We first show that δ is well-defined, i.e., that the definition does not depend on the choice of the member of the equivalence class. Let $u\ R_f\ v$. Then, since R_f is right invariant we have $u \cdot a\ R_f\ v \cdot a$ for any $a \in \Sigma$. Hence $\delta([u]_{R_f}, a) = [u \cdot a]_{R_f} = [v \cdot a]_{R_f}$.

Clearly by definition δ is a total function with domain $(\Sigma^*/R_f) \times \Sigma$.

To show that λ is well-defined we first have to see that if $u \cdot a \in Pref(dom(f))$ then $lco_f(u \cdot a)$ is a prefix of $lco_f(u)$.
Indeed, since $\{w \in \Sigma^* \mid u \cdot w \in dom(f)\} \supseteq \{w \in \Sigma^* \mid u \cdot a \cdot w \in dom(f)\}$ we have that $\{f(u \cdot w) \mid w \in \Sigma^*, u \cdot w \in dom(f)\} \supseteq \{f(u \cdot a \cdot w \mid w \in \Sigma^*, u \cdot a \cdot w \in dom(f)\}$ and therefore $lco_f(u \cdot a) = \bigwedge\{f(u \cdot a \cdot w \mid w \in \Sigma^*, u \cdot a \cdot w \in dom(f)\}$ is a prefix of $lco_f(u) = \bigwedge\{f(u \cdot w) \mid w \in \Sigma^*, u \cdot w \in dom(f)\}$.

We now show that the definition of λ does not depend on the choice of the member of the equivalence class. Let $v \in [u]_{R_f}$. Since $u\ R_f\ v$ we have $u \cdot a\ R_f\ v \cdot a$. If $u \cdot a \notin Pref(dom(f))$ then by the definition of R_f we have $v \cdot a \notin Pref(dom(f))$ and therefore $\lambda([u], a) = \lambda([v], a) = \varepsilon$.

If $u \cdot a \in Pref(dom(f))$ then there exist $u', v' \in \Sigma'^*$ such that for all $w \in \Sigma^*$ with $u \cdot w \in dom(f)$ we have

$$u'^{-1} f(u \cdot w) = v'^{-1} f(v \cdot w).$$

This implies that

$$\begin{aligned}
\bigwedge_{w \in \Sigma^*, u \cdot w \in dom(f)} u'^{-1} f(u \cdot w) &= \bigwedge_{w \in \Sigma^*, v \cdot w \in dom(f)} v'^{-1} f(v \cdot w) \\
u'^{-1} \bigwedge_{w \in \Sigma^*, u \cdot w \in dom(f)} f(u \cdot w) &= v'^{-1} \bigwedge_{w \in \Sigma^*, v \cdot w \in dom(f)} f(v \cdot w) \\
u'^{-1} lco_f(u) &= v'^{-1} lco_f(v).
\end{aligned}$$

Similarly for all $w \in \Sigma^*$ such that $u \cdot a \cdot w \in dom(f)$ we have

$$u'^{-1} f(u \cdot a \cdot w) = v'^{-1} f(v \cdot a \cdot w).$$

This implies that

$$\begin{aligned}
\bigwedge_{w \in \Sigma^*, u \cdot a \cdot w \in dom(f)} u'^{-1} f(u \cdot a \cdot w) &= \bigwedge_{w \in \Sigma^*, v \cdot a \cdot w \in dom(f)} v'^{-1} f(v \cdot a \cdot w) \\
u'^{-1} \bigwedge_{w \in \Sigma^* . u \cdot a \cdot w \in dom(f)} f(u \cdot a \cdot w) &= v'^{-1} \bigwedge_{w \in \Sigma^*, v \cdot a \cdot w \in dom(f)} f(v \cdot a \cdot w) \\
u'^{-1} lco_f(u \cdot a) &= v'^{-1} lco_f(v \cdot a).
\end{aligned}$$

Hence

$$lco_f(u)^{-1} lco_f(u \cdot a) = [u' v'^{-1} lco_f(v)]^{-1} (u' v'^{-1} lco_f(v \cdot a)) = lco_f(v)^{-1} lco_f(v \cdot a)$$

and therefore the definition of λ is correct.

Now we show that the definition of Ψ does not depend on the choice of the member of the equivalence class. Let $u\ R_f\ v$ and $u \in dom(f)$. Hence there exist $u', v' \in \Sigma'^*$ such that (for $w = \varepsilon$)

$$u'^{-1} f(u) = v'^{-1} f(v)$$

and thus

$$u'^{-1} lco_f(u)\ (lco_f(u)^{-1} f(u)) = v'^{-1} lco_f(v)\ (lco_f(v)^{-1} f(v)).$$

Since $u'^{-1} lco_f(u) = v'^{-1} lco_f(v)$ we obtain

$$lco_f(u)^{-1} f(u) = lco_f(v)^{-1} f(v).$$

To prove that $O_{\mathcal{T}_f} = f$ we first show by induction that for $u \in Pref(dom(f))$ we have $\iota \cdot \lambda^*([\varepsilon], u) = lco_f(u)$. First, for $u = \varepsilon$ we have $\iota \cdot \lambda^*([\varepsilon], \varepsilon) = \iota = lco_f(\varepsilon)$. For the induction step assume $\iota \cdot \lambda^*([\varepsilon], u) = lco_f(u)$ and let $a \in \Sigma$. Then

$$\begin{aligned} \iota \cdot \lambda^*([\varepsilon], u \cdot a) &= \iota \cdot \lambda^*([\varepsilon], u) \cdot \lambda([u], a) = lco_f(u) \cdot \lambda([u], a) \\ &= lco_f(u)(lco_f(u)^{-1} lco_f(u \cdot a)) = lco_f(u \cdot a). \end{aligned}$$

For $u \in dom(f)$ we have

$$O_{\mathcal{T}_f}(u) = \iota \cdot \lambda^*([\varepsilon], u) \cdot \Psi([u]) = lco_f(u) \cdot \Psi([u]) = lco_f(u)(lco_f(u)^{-1} f(u)) = f(u).$$

□

Definition 5.4.9 The transducer $\mathcal{T}_f$ in Proposition 5.4.8 is called the *Myhill-Nerode transducer* for f.

Similarly as with the deterministic finite-state automata we obtained another characterization of the class of functions represented by subsequential transducers.

Theorem 5.4.10 *Let $f : \Sigma^* \to \Sigma'^*$ be a function. Then f is the output function of a subsequential finite-state transducer iff the index of R_f is finite.*

Theorem 5.4.11 *Let $f : \Sigma^* \to \Sigma'^*$ be a function such that the index of R_f is finite. Then the Myhill-Nerode transducer $\mathcal{T}_f$ for f is minimal with respect to number of states among all subsequential classical finite-state transducers with initial output and total transition function representing f.*

Proof. Let $\mathcal{T}$ be a subsequential classical finite-state transducers with initial output and total transition function representing f. The definition of $R_{\mathcal{T}}$ (cf. Proposition 5.4.6) implies that the number of equivalence classes of $R_{\mathcal{T}}$ is bounded by the number of states $|Q|$ of $\mathcal{T}$. Corollary 5.4.7 shows that the number of equivalence classes of the Nerode equivalence relation R_f, which coincides with the number of states of $\mathcal{T}_f$, does not exceed $|Q|$. □

Remark 5.4.12 Note that there can be many equivalent non-isomorphic minimal subsequential finite-state transducers. The topology of any of the minimal transducers is isomorphic to the Myhill-Nerode transducer, but some parts of the outputs can be moved along the paths.

The following generalises the minimality result for subsequential finite-state transducers with *arbitrary* transition function.

Lemma 5.4.13 *Let $f : \Sigma^* \to \Sigma'^*$ be a function such that the index of the Myhill-Nerode relation R_f is finite. Then the Myhill-Nerode transducer $\mathcal{T}_f$ has at most one state p such that no final state is reachable from p. All states are reachable from the start state.*

Proof. Clearly a state $[u]_{R_f}$ is reached from the start state with input u. Assume that $\mathcal{T}_f$ has states $p = [v]_{R_f}$ and $p' = [v']_{R_f}$ such that no final state can be reached from p or p'. Then v and v' cannot be extended to strings in the domain of f. The definition of the Myhill-Nerode relation shows that $v \; R_f \; v'$, which implies $p = p'$. □

For the rest of this chapter we use the notation $\mathcal{T}'_f$ for the transducer obtained from the Myhill-Nerode-Transducer $\mathcal{T}_f$ via trimming. We call $\mathcal{T}'_f$ the *trimmed Myhill-Nerode transducer* for f. Obviously $\mathcal{T}_f$ and $\mathcal{T}'_f$ are equivalent.

Theorem 5.4.14 *Let $f : \Sigma^* \to \Sigma'^*$ be a function such that the index of the Myhill-Nerode relation R_f is finite. Then the trimmed Myhill-Nerode transducer $\mathcal{T}'_f$ has the minimal number of states among all classical subsequential transducers with initial output representing f.*

Proof. First note that it does not make a difference if we refer to all trimmed or all classical subsequential transducers with initial output and partial transition function representing f: if a transducer is not trimmed we can always delete a state, again obtaining a smaller transducer representing f. Assume there exists a trimmed classical subsequential transducer $\mathcal{T}$ with initial output representing f with a smaller number of states than $\mathcal{T}'_f$.
Case 1: The transition function of $\mathcal{T}$ is total. In this case according to Theorem 5.4.11 the number of states of $\mathcal{T}$ is greater or equal to the number of states of $\mathcal{T}_f$ which is greater or equal to the number of states of $\mathcal{T}'_f$ (Lemma 5.4.13). This yields a contradiction.
Case 2: The transition function of $\mathcal{T}$ is not total. In this case $\mathit{Pref}(\mathit{dom}(f)) \neq \Sigma^*$ and therefore there exist a state in $\mathcal{T}_f$ from which no final state can be reached. Adding a single new state plus new transitions in the obvious way we obtain an equivalent classical subsequential transducer $\mathcal{T}'$ with initial output and *total* transition function. Let n_f, n'_f, n and n' respectively denote the number of states of $\mathcal{T}_f$, $\mathcal{T}'_f$, $\mathcal{T}$, and $\mathcal{T}'$. We have $n' - 1 = n$ and $n_f - 1 = n'_f$. From the assumption we have $n < n'_f$ and therefore $n' < n_f$. Since the transducer $\mathcal{T}'$ has a total transition function this contradicts Theorem 5.4.11 □

5.5 Minimization of subsequential transducers

In this section we show how to minimize a given subsequential finite-state transducer, essentially following the approach in [Mohri, 2000]. We present the procedure for the case of classical subsequential finite-state transducers. But essentially the same technique can be used for other target monoids that satisfy some additional conditions [Gerdjikov and Mihov, 2017a]. In the first theoretical part we show that in order to obtain a minimal subsequential finite-state transducer from a given subsequential input transducer it suffices to compute an equivalent pseudo-minimal (cf. Section 3.7) transducer in a special "canonical form" to be defined below. Afterwards we describe the missing details of the two algorithmic steps needed, showing (1) how to convert the input transducer to canonical form and (2) how to apply pseudo-minimization.

For the minimization of subsequential transducers we assume that input transducers are always trimmed. This is motivated by the fact that some of the following definitions only make sense for trimmed transducers.

Theoretical background

We start with the first step of the minimization procedure mentioned above. Let us remind that the output function of a state q is defined (cf. Definition 5.1.7) as

$$O^q_{\mathcal{T}}(\alpha) := \lambda^*(q,\alpha) \cdot \Psi(\delta^*(q,\alpha))$$

for α such that $\delta^*(q,\alpha)$ is final. For a trimmed transducer the domain of $O^q_{\mathcal{T}}$ is non-empty

Definition 5.5.1 Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed classical subsequential finite-state transducer with initial output. The *maximal state output* $mso_{\mathcal{T}}(q)$ for a state $q \in Q$ is defined as

$$mso_{\mathcal{T}}(q) := \bigwedge_{w \in \Sigma^* \ \&\ \delta^*(q,w) \in F} O^q_{\mathcal{T}}(w).$$

Note that $mso_{\mathcal{T}} : Q \to \Sigma'^*$ is a total function.

Definition 5.5.2 Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed classical subsequential finite-state transducer with initial output. Then the *canonical form* of $\mathcal{T}$ is the subsequential finite-state transducer with initial output

$$\mathcal{T}' := \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda', \iota', \Psi' \rangle$$

where

- $\iota' := \iota \cdot mso_{\mathcal{T}}(q_0)$,
- $\Psi'(q) := mso_{\mathcal{T}}(q)^{-1}\Psi(q)$ for all $q \in F$,
- $\lambda'(q,\sigma) := mso_{\mathcal{T}}(q)^{-1}(\lambda(q,\sigma) \cdot mso_{\mathcal{T}}(\delta(q,\sigma)))$, for all $q \in Q, \sigma \in \Sigma$ such that $\delta(q,\sigma)$ is defined.

The definition ensures that the maximal state output of a state is now already produced when reaching the state. For example, consider the third clause where we define the output of a σ-transition leading from q to $p = \delta(q, \sigma)$. We do not only produce $\lambda(q, \sigma)$ but also the maximal state output for p. The correcting factor $mso_{\mathcal{T}}(q)^{-1}$, which deletes a prefix of $\lambda(q, \sigma) \cdot mso_{\mathcal{T}}(p)$, takes into account that when reaching q in the new transducer $mso_{\mathcal{T}}(q)$ has already been produced.

Definition 5.5.3 A trimmed classical subsequential finite-state transducer with initial output $\mathcal{T}$ is in *canonical form* if $\mathcal{T}$ is isomorphic to its canonical form $\mathcal{T}'$.

Corollary 5.5.4 *A trimmed classical subsequential finite-state transducer with initial output $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ is in canonical form iff for every state $q \in Q$ we have $mso(q) = \varepsilon$.*

Proof. Assume that for every state $q \in Q$ we have $mso(q) = \varepsilon$. Then by Definition 5.5.2 $\iota' = \iota$, $\Psi' = \Psi$ and $\lambda' = \lambda$ and therefore $\mathcal{T}' = \mathcal{T}$.
Assume now that $\mathcal{T}' = \mathcal{T}$. Since $\iota' = \iota$ from Definition 5.5.2 we have $mso(q_0) = \varepsilon$. Since $\lambda' = \lambda$ by induction on the depth of q we show that $mso(q) = \varepsilon$. □

Example 5.5.5 See Figure 5.5 and the explanation in Example 5.5.17 for an illustration of the canonical form.

The following proposition shows that conversion to canonical form leads to an equivalent trimmed transducer.

Proposition 5.5.6 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed classical subsequential finite-state transducer with initial output and let*

$$\mathcal{T}' = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda', \iota', \Psi' \rangle$$

be the canonical form of $\mathcal{T}$. Then $\mathcal{T}'$ is trimmed and equivalent to $\mathcal{T}$.

Proof. Clearly $dom(O_{\mathcal{T}}) = dom(O_{\mathcal{T}'})$. Since we do not modify the transition function and the set of final states also $\mathcal{T}'$ is trimmed. By definition we have $\iota' := \iota \cdot mso_{\mathcal{T}}(q_0)$ and thus

$$\begin{aligned} \iota' \cdot \lambda'(q_0, \sigma) &= \iota \cdot mso_{\mathcal{T}}(q_0) \cdot (mso_{\mathcal{T}}(q_0)^{-1}(\lambda(q_0, \sigma) \cdot mso_{\mathcal{T}}(\delta(q), \sigma))) \\ &= \iota \cdot \lambda(q_0, \sigma) \cdot mso_{\mathcal{T}}(\delta(q_0, \sigma)). \end{aligned}$$

A simple induction shows that for any $u \in \Sigma^*$ such that $\delta^*(q_0, u)$ is defined we have

$$\iota' \cdot \lambda'^*(q_0, u) = \iota \cdot \lambda^*(q_0, u) \cdot mso_{\mathcal{T}}(\delta^*(q_0, u)).$$

Now let $u \in dom(O_{\mathcal{T}})$. Then

$$\begin{aligned} O_{\mathcal{T}}(u) &= \iota' \cdot \lambda'^*(q_0, u) \cdot \Psi'(\delta^*(q_0, u)) \\ &= \iota \cdot \lambda^*(q_0, u) \cdot mso_{\mathcal{T}}(\delta^*(q_0, u)) \cdot (mso_{\mathcal{T}}(\delta^*(q_0, u))^{-1} \Psi(\delta^*(q_0, u))) \\ &= \iota \cdot \lambda^*(q_0, u) \cdot \Psi(\delta^*(q_0, u)) \\ &= O_{\mathcal{T}'}(u). \end{aligned}$$

□

Corollary 5.5.7 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed classical subsequential finite-state transducer with initial output representing the function f and let $\mathcal{T}' = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda', \iota', \Psi' \rangle$ be the canonical form of $\mathcal{T}$. Let $u \in \Sigma^*$ be a word such that $\delta^*(q_0, u)$ is defined. Then*

$$\iota' \cdot \lambda'^*(q_0, u) = lco_f(u).$$

Proof. Let $q := \delta^*(q_0, u)$. Since each state is on a successful path there exists at least one string $w \in \Sigma^*$ such that $u \cdot w \in dom(f)$. Therefore

$$\begin{aligned}
lco_f(u) &= \bigwedge_{w \in \Sigma^* \ \&\ u \cdot w \in dom(f)} f(u \cdot w) \\
&= \bigwedge_{w \in \Sigma^* \ \&\ u \cdot w \in dom(f)} \iota \cdot \lambda^*(q_0, u) \cdot \lambda^*(q, w) \cdot \Psi(\delta^*(q, w)) \\
&= \iota \cdot \lambda^*(q_0, u) \cdot \bigwedge_{w \in \Sigma^* \ \&\ u \cdot w \in dom(f)} \lambda^*(q, w) \cdot \Psi(\delta^*(q, w)) \\
&= \iota \cdot \lambda^*(q_0, u) \cdot \bigwedge_{w \in \Sigma^* \ \&\ u \cdot w \in dom(f)} O^q_{\mathcal{T}}(w) \\
&= \iota \cdot \lambda^*(q_0, u) \cdot mso_{\mathcal{T}}(q) \\
&= \iota' \cdot \lambda'^*(q_0, u).
\end{aligned}$$

□

Proposition 5.5.8 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed classical subsequential transducer in canonical form representing the function f. Let $q_1, q_2 \in Q$ be two states. Then q_1 and q_2 are equivalent iff*

1. *$q_1 \in F$ iff $q_2 \in F$,*
2. *if $q_1 \in F$, then $\Psi(q_1) = \Psi(q_2)$,*
3. *for all $\sigma \in \Sigma$: $\delta(q_1, \sigma)$ is defined iff $\delta(q_2, \sigma)$ is defined. If both are defined, then $\delta(q_1, \sigma)$ and $\delta(q_2, \sigma)$ are equivalent and $\lambda(q_1, \sigma) = \lambda(q_2, \sigma)$.*

Proof. Obviously, if the three conditions are satisfied, then q_1 and q_2 are equivalent. To prove the converse direction let q_1 and q_2 be equivalent. If $q_1 \in F$, then $O^{q_1}_{\mathcal{T}}(\varepsilon) = \Psi(q_1)$ is defined. Since $O^{q_1}_{\mathcal{T}} = O^{q_2}_{\mathcal{T}}$ it follows that $O^{q_2}_{\mathcal{T}}(\varepsilon)$ is defined and $O^{q_1}_{\mathcal{T}}(\varepsilon) = O^{q_2}_{\mathcal{T}}(\varepsilon) = \Psi(q_2)$. Properties 1 and 2 follow.

Assume that $\delta(q_1, \sigma)$ is defined. Then σ is the first letter of a string $w = \sigma w'$ in the domain of $O^{q_1}_{\mathcal{T}} = O^{q_2}_{\mathcal{T}}$, which shows that $\delta(q_2, \sigma)$ is defined.

Let $u, v \in \Sigma^*$ be two words such that $\delta^*(q_0, u) = q_1$ and $\delta^*(q_0, v) = q_2$. From Corollary 5.5.7 we get

$$\begin{aligned}
\iota \cdot \lambda^*(q_0, u \cdot \sigma) &= lco_f(u \cdot \sigma) \\
\iota \cdot \lambda^*(q_0, u) &= lco_f(u).
\end{aligned}$$

It follows that

$$\lambda(q_1, \sigma) = lco_f(u)^{-1} lco_f(u \cdot \sigma).$$

In the same way we obtain

$$\lambda(q_2, \sigma) = lco_f(v)^{-1} lco_f(v \cdot \sigma).$$

Since $O_{\mathcal{T}}^{q_1} = O_{\mathcal{T}}^{q_2}$ it follows that $u \; R_f \; v$. As we have seen in the proof of Proposition 5.4.8 this implies that $lco_f(u)^{-1} lco_f(u \cdot \sigma) = lco_f(v)^{-1} lco_f(v \cdot \sigma)$ and we obtain

$$\lambda(q_1, \sigma) = lco_f(u)^{-1} lco_f(u \cdot \sigma) = lco_f(v)^{-1} lco_f(v \cdot \sigma) = \lambda(q_2, \sigma).$$

For every $w \in \Sigma^*$ such that $\delta^*(q_1, \sigma \cdot w) \in F$, using $O_{\mathcal{T}}^{q_1} = O_{\mathcal{T}}^{q_2}$ we see that

$$O_{\mathcal{T}}^{q_1}(\sigma \cdot w) = \lambda(q_1, \sigma) \cdot O_{\mathcal{T}}^{\delta(q_1,\sigma)}(w) = \lambda(q_2, \sigma) \cdot O_{\mathcal{T}}^{\delta(q_2,\sigma)}(w) = O_{\mathcal{T}}^{q_2}(\sigma \cdot w).$$

Hence we obtain $O_{\mathcal{T}}^{\delta(q_1,\sigma)} = O_{\mathcal{T}}^{\delta(q_2,\sigma)}$. □

The next lemma shows that for a classical subsequential finite-state transducer with initial output in canonical form, Myhill-Nerode equivalence of two words boils down to equivalence of the states reached.

Lemma 5.5.9 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed classical subsequential finite-state transducer with initial output in canonical form representing the function $f : \Sigma^* \to \Sigma'^*$. Let $u, v \in \Sigma^*$, let $\delta^*(q_0, u) = q_1$ and $\delta^*(q_0, v) = q_2$ be defined. Then $u \; R_f \; v$ iff q_1 is equivalent to q_2.*

Proof. (“⇒”) Clearly, since $u \; R_f \; v$ we have $dom(O_{\mathcal{T}}^{q_1}) = dom(O_{\mathcal{T}}^{q_2})$. In the proof of Proposition 5.4.8 we have seen that $u'^{-1} lco_f(u) = v'^{-1} lco_f(v)$. Let $w \in \Sigma^*$ such that $u \cdot w \in dom(f)$. Let u' and v' as in Definition 5.4.1. Using Corollary 5.5.7 and the above equation we get

$$\begin{aligned}
u'^{-1} f(u \cdot w) &= v'^{-1} f(v \cdot w) \\
u'^{-1} lco_f(u) \cdot (lco_f(u)^{-1} f(u \cdot w)) &= v'^{-1} lco_f(v) \cdot (lco_f(v)^{-1} f(v \cdot w)) \\
lco_f(u)^{-1} f(u \cdot w) &= lco_f(v)^{-1} f(v \cdot w) \\
(\iota \cdot \lambda^*(q_0, u))^{-1} f(u \cdot w) &= (\iota \cdot \lambda^*(q_0, v))^{-1} f(v \cdot w).
\end{aligned}$$

We obtain

$$\begin{aligned}
& [\iota \cdot \lambda^*(q_0, u)]^{-1} (\iota \cdot \lambda^*(q_0, u) \cdot \lambda^*(q_1, w) \cdot \Psi(\delta^*(q_1, w))) \\
= \quad & [\iota \cdot \lambda^*(q_0, v)]^{-1} (\iota \cdot \lambda^*(q_0, v) \cdot \lambda^*(q_2, w) \cdot \Psi(\delta^*(q_2, w)))
\end{aligned}$$

and thus $\lambda^*(q_1, w) \cdot \Psi(\delta^*(q_1, w)) = \lambda^*(q_2, w) \cdot \Psi(\delta^*(q_2, w))$. Hence for every $w \in dom(O_{\mathcal{T}}^{q_1})$ we have $O_{\mathcal{T}}^{q_1}(w) = O_{\mathcal{T}}^{q_2}(w)$, which shows that q_1 and q_2 are equivalent.

(“⇐”) Let q_1 be equivalent to q_2. Let $u' := \iota \cdot \lambda^*(q_0, u)$ and $v' := \iota \cdot \lambda^*(q_0, v)$. Let $w \in \Sigma^*$ be a word such that $u \cdot w \in dom(f)$. Then

$$\begin{aligned}
u'^{-1} f(u \cdot w) &= [\iota \cdot \lambda^*(q_0, u)]^{-1} (\iota \cdot \lambda^*(q_0, u) \cdot \lambda^*(q_1, w) \cdot \Psi(\delta^*(q_1, w))) \\
&= \lambda^*(q_1, w) \cdot \Psi(\delta^*(q_1, w)) \\
&= \lambda^*(q_2, w) \cdot \Psi(\delta^*(q_2, w)) \\
&= [\iota \cdot \lambda^*(q_0, v)]^{-1} (\iota \cdot \lambda^*(q_0, v) \cdot \lambda^*(q_2, w) \cdot \Psi(\delta^*(q_2, w))) \\
&= v'^{-1} f(v \cdot w).
\end{aligned}$$

□

Theorem 5.5.10 *A classical subsequential finite-state transducer $\mathcal{T}$ with initial output in canonical form representing the string function f is minimal (in terms of the number of states) among all classical subsequential finite-state transducers with initial output representing f iff $\mathcal{T}$ is trimmed and there are no distinct equivalent states in $\mathcal{T}$.*

Proof. Clearly $\mathcal{T}$ is trimmed since otherwise we could build an equivalent transducer with a smaller number of states. Let Q and Q'_f respectively denote the set of states of $\mathcal{T}$ and the trimmed Myhill-Nerode transducer $\mathcal{T}'_f$ for f. For each state $q \in Q$ there exists a string $u_q \in \Sigma^*$ such that q is reached in $\mathcal{T}$ from the start state with input u_q. It follows from Proposition 5.4.6 that the mapping $\chi : q \mapsto [u_q]_{R_f}$ is well-defined. Since in $\mathcal{T}$ from each state q a final state can be reached we have $\chi(q) \in Q'_f$ for all $q \in Q$. On the other hand if $[u]_{R_f}$ is a state in Q'_f, then u is a prefix of a string in the domain of f. It follows that also in $\mathcal{T}$ a state q is reached from the start with in input u and we have $\chi(q) = [u]_{R_f}$. Hence $\chi : Q \to Q'_f$ is a surjective mapping.

Now assume that $\mathcal{T}$ is minimal in the above sense and $\mathcal{T}$ has distinct states p and p' that are equivalent, respectively reached from the start state with input strings u and u'. Lemma 5.5.9 implies that $u \; R_f u'$. Then $[u]_{R_f} = [u']_{R_f}$ and the mapping χ is not injective. This would mean that the number of states of $\mathcal{T}$ is greater than the number of states of $\mathcal{T}'_f$, which contradicts the minimality of $\mathcal{T}$.

Conversely assume that $\mathcal{T}$ is trimmed and there are no distinct equivalent states in $\mathcal{T}$. Using Proposition 5.4.6 and Lemma 5.5.9 it follows that χ is a bijection. It follows from Theorem 5.4.14 that $\mathcal{T}$ is minimal in terms of the number of states among all classical subsequential finite-state transducers with initial output representing f. □

Conversion to canonical form - computing maximal state outputs

The conversion to canonical form requires finding the maximal state output $mso_{\mathcal{T}}$ for each state of the transducer. We use the following notions and constructions.

Definition 5.5.11 Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a classical subsequential finite-state transducer with initial output. Then the monoidal finite-state automaton

$$\mathcal{A}_{\mathcal{T}} := \langle \Sigma'^*, Q \cup \{f\}, \{q_0\}, \{f\}, \Delta \rangle$$

where $f \notin Q$ is a new state and

$$\Delta = \{\langle q, \Psi(q), f\rangle \mid q \in F\} \cup \{\langle q', \lambda(q', \sigma), q''\rangle \mid \langle q', \sigma, q''\rangle \in \delta\}$$

is called the *output automaton* of $\mathcal{T}$.

Clearly, the language of $\mathcal{A}_{\mathcal{T}}$ is equal to $\iota^{-1} codom(O_{\mathcal{T}})$ and for each state $q \in Q$ we have

$$L_{\mathcal{A}_{\mathcal{T}}}(q) = \bigcup_{w \in \Sigma^* \;\&\; \delta^*(q,w) \in F} O^q_{\mathcal{T}}(w).$$

Following Proposition 3.2.1 we compute a classical finite-state automaton $\mathcal{A}'_{\mathcal{T}}$ with an extended set of states Q' without ε-transitions such that for each $q \in Q$ we have $L_{\mathcal{A}_{\mathcal{T}}}(q) = L_{\mathcal{A}'_{\mathcal{T}}}(q)$. $\mathcal{A}'_{\mathcal{T}}$ is called the *expanded output automaton* of $\mathcal{T}$.

Example 5.5.12 See Figure 5.5 and the explanation in Example 5.5.17 for an example output automaton and the corresponding expanded output automaton.

Proposition 5.5.13 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a classical subsequential finite-state transducer with initial output, let*

$$\mathcal{A}'_{\mathcal{T}} = \langle \Sigma', Q \cup Q'' \cup \{f\}, q_0, F'', \Delta'' \rangle$$

be the expanded output automaton of $\mathcal{T}$, let $q \in Q$. Then

$$L_{\mathcal{A}'_{\mathcal{T}}}(q) = \bigcup_{w \in \Sigma^*, \delta^*(q,w) \in F} O^q_{\mathcal{T}}(w)$$

and

$$mso_{\mathcal{T}}(q) = \bigwedge L_{\mathcal{A}'_{\mathcal{T}}}(q).$$

The following proposition facilitates the effective calculation of the longest common prefix of an automaton language.

Proposition 5.5.14 *Let $\mathcal{A} = \langle \Sigma', Q, q_0, F, \delta \rangle$ be a trimmed deterministic finite-state automaton. Let*

$$\pi = q_0 \to^{a_1} q_1 \to^{a_2} q_2 \dots q_{k-1} \to^{a_k} q_k$$

be a path starting at q_0. Then $w = \bigwedge L(\mathcal{A})$ for $w = a_1 a_2 \dots a_k$ iff the following properties hold:

- *q_k is final or there are more than one outgoing transitions from q_k. i.e.*

$$q_k \in F \vee |\{\sigma \in \Sigma' \mid !\delta(q_k, \sigma)\}| > 1.$$

- *For each i in $0, \dots, k-1$ the state q_i is not final and there is exactly one outgoing transition from q_i. I.e.*

$$\forall i \in \{0, \dots, k-1\} : q_i \notin F \ \& \ |\{\sigma \in \Sigma' \mid !\delta(q_i, \sigma)\}| = 1.$$

If the conditions are satisfied, the path π is called the *maximal unique path* of $\mathcal{A}$. Now in order to calculate $\bigwedge L_{\mathcal{A}'_{\mathcal{T}}}(q)$ we will proceed with determinizations for $\mathcal{A}'_{\mathcal{T}}$, using distinct start states.

Corollary 5.5.15 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed classical subsequential finite-state transducer with initial output and let*

$$\mathcal{A}'_{\mathcal{T}} = \langle \Sigma', Q \cup Q'' \cup \{f\}, q_0, F'', \Delta'' \rangle$$

be the expanded output automaton of $\mathcal{T}$. Let $\mathcal{D}^q = \langle \Sigma', Q_q, \{q\}, F_q, \delta_q \rangle$ be the deterministic finite-state automaton obtained from the determinization of ${\mathcal{A}'_{\mathcal{T}}}^q = \langle \Sigma', Q \cup Q'' \cup \{f\}, q, F'', \Delta'' \rangle$. Then

$$mso_{\mathcal{T}}(q) = \bigwedge L(\mathcal{D}^q) = w_q,$$

where $w_q \in \Sigma'^*$ is the label of the maximal unique path of $\mathcal{D}^q$.

Remark 5.5.16 In order to find the longest common prefix of the language of ${\mathcal{A}'_{\mathcal{T}}}^{q}$ (cf. Corollary 5.5.15) we can proceed by determinizing only the initial part of the automaton, until we reach the state q_k for which the conditions in Proposition 5.5.14 are fulfilled. Hence only a small part of the automaton has to be determinized. The complexity of finding $mso_{\mathcal{T}}(q)$ is $O(|w_q||Q \cup Q''|^2)$ (cf. Program 8.3.5).

Example 5.5.17 All steps of the minimization of a subsequential transducer are illustrated in Figure 5.5. The first graph represents a subsequential transducer $\mathcal{T}$. Below the output automaton and the expanded output automaton for $\mathcal{T}$ are shown. Below maximal unique paths after determinization with distinct start states $1, \ldots, 6$ are shown. From these values we obtain the longest common output for each state of $\mathcal{T}$. Having the longest common outputs we can compute a transducer $\mathcal{T}'$ with initial output in canonical form. The last step is the pseudo-determinization of $\mathcal{T}'$, we obtain the minimal transducer $\mathcal{T}''$ at the bottom.

As a resume, in order to compute the canonical subsequential transducer for a given subsequential transducer $\mathcal{T}$ we first compute the maximal state output for each state using Corollary 5.5.15. Having the maximal state outputs we convert $\mathcal{T}$ to canonical form $\mathcal{T}'$ by modifying outputs following Definition 5.5.2.

Pseudo-minimization - computing the minimal subsequential transducer

In order to construct the minimal transducer from a canonical trimmed subsequential transducer we apply a pseudo-minimization procedure as in Section 3.7 combined with the colouring of the final states (cf. Section 3.6) induced by the state output function Ψ. The following proposition formalizes this idea.

Proposition 5.5.18 *Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a canonical trimmed subsequential finite-state transducer with initial output. Consider the new alphabet*

$$\Gamma := \{\langle c, \lambda(q,c)\rangle \mid c \in \Sigma, q \in Q : !\delta(q,c)\},$$

and the set of transitions $\delta_{\mathcal{A}} := \{\langle q, \langle c, \lambda(q,c)\rangle, q'\rangle \mid \langle q, c, q'\rangle \in \delta\}$. Then

$$\mathcal{A}_{\mathcal{T}} := \langle \Gamma, codom(\Psi), Q, q_0, F, \delta_{\mathcal{A}}, \Psi \rangle$$

is a $codom(\Psi)$-coloured deterministic finite-state automaton. Let

$$\mathcal{A}'_{\mathcal{T}} = \langle \Gamma, codom(\Psi), Q', q'_0, F', \delta'_{\mathcal{A}}, \Psi' \rangle$$

denote the minimal $codom(\Psi)$-coloured deterministic finite-state automaton equivalent to $\mathcal{A}_{\mathcal{T}}$. Then the subsequential finite-state transducer

$$\mathcal{T}' := \langle \Sigma, \Sigma', Q', q'_0, F', \delta', \lambda', \iota, \Psi' \rangle,$$

where

$$\begin{aligned} \delta' &:= \{\langle q, c, q'\rangle \mid \langle q, \langle c, \alpha\rangle, q'\rangle \in \delta'_{\mathcal{A}}\} \\ \lambda' &:= \{\langle q, c, \alpha\rangle \mid \langle q, \langle c, \alpha\rangle, q'\rangle \in \delta'_{\mathcal{A}}\} \end{aligned}$$

is the minimal canonical subsequential finite-state transducer equivalent to $\mathcal{T}$.

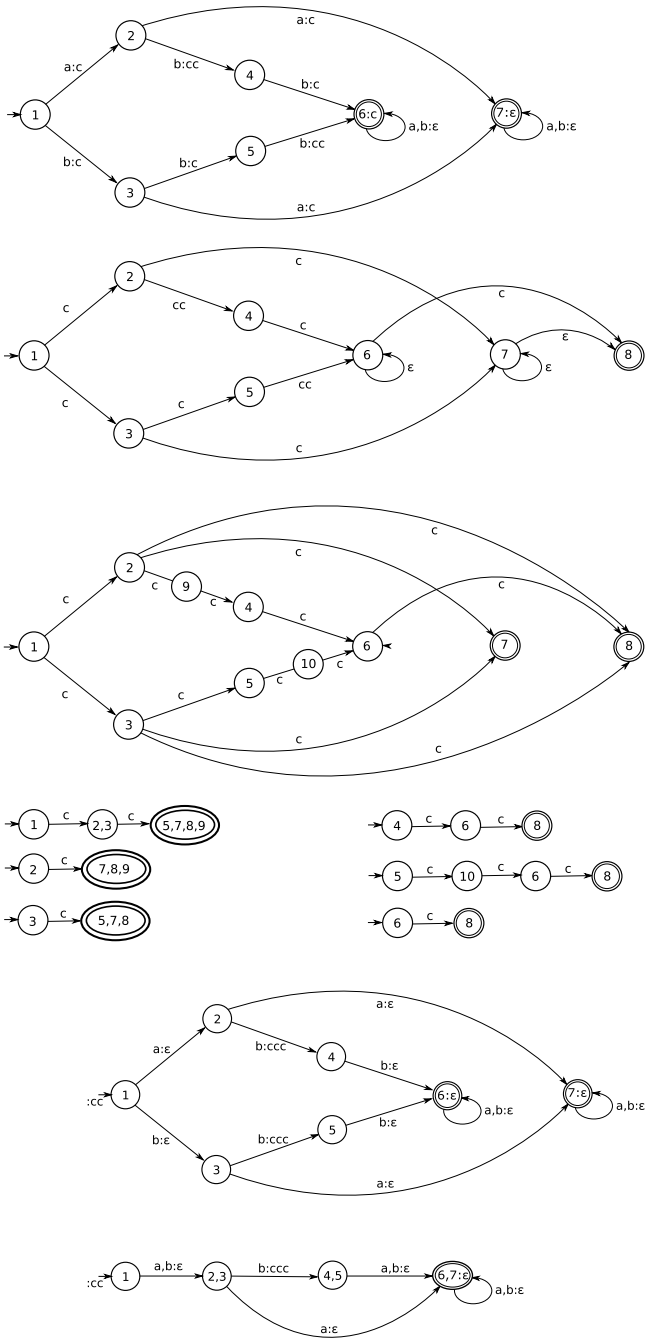

Figure 5.5: Minimization of a subsequential transducer, cf. Example 5.5.17.

The computation of classes of equivalent states of the coloured deterministic finite-state automaton $\mathcal{A}_{\mathcal{T}}$ is based on the relations R_i $(i \geq 0)$, which were defined in the following way (cf. Remark 3.5.8):

$$\begin{array}{rcl} q\ R_0\ p & \leftrightarrow & (q \in F \leftrightarrow p \in F)\ \&\ (q \in F \rightarrow \Psi(q) = \Psi(p)) \\ q\ R_{i+1}\ p & \leftrightarrow & q\ R_i\ p\ \&\ \forall \bar{a} \in \Gamma : \delta_{\mathcal{A}}(q,\bar{a}), \delta_{\mathcal{A}}(p,\bar{a}) \text{ undefined, or} \\ & & \delta_{\mathcal{A}}(q,\bar{a})\ R_i\ \delta_{\mathcal{A}}(p,\bar{a}). \end{array}$$

Corollary 5.5.19 *The equivalence relation for minimizing the canonical subsequential finite-state transducer $\mathcal{T}$ coincides with the relation $R = \bigcap_0^\infty R_i$ where*

1. $R_0 = ker_Q(f)$
2. $R_{i+1} = \bigcap_{a\in\Gamma} ker_Q(f_{\bar{a}}^{(i)}) \cap R_i$

For $a \in \Gamma$ and $i \in \mathbb{N}$ the function $f_{\bar{a}}^{(i)} : Q \rightarrow (Q/R_i) \cup \{\perp\}$ is defined as

$$f_{\bar{a}}^{(i)}(q) := \begin{cases} \perp & \text{if } \delta_{\mathcal{A}}(q,\bar{a}) \text{ undefined,} \\ [\delta_{\mathcal{A}}(q,\bar{a})]_{R_i} & \text{otherwise.} \end{cases}$$

and $f : Q \rightarrow codom(\Psi) \cup \{0_c\}$ is defined as

$$f(q) := \begin{cases} \Psi(q) & \text{if } q \in F, \\ 0_c & \text{otherwise.} \end{cases}$$

5.6 Numerical subsequential transducers

In this section we show how the results from the previous sections can be transferred to the monoid of natural numbers.

Definition 5.6.1 Let $\mathcal{T} = \langle \Sigma, \Sigma', Q, q_0, F, \delta, \lambda, \Psi \rangle$ be a classical subsequential transducer. Let $\phi : \Sigma'^* \rightarrow \mathcal{M}$ be a homomorphism between the monoids Σ'^* and $\mathcal{M} = \langle M, \bullet, e \rangle$. Then the monoidal subsequential transducer $\mathcal{T}_\phi = \langle \Sigma, \mathcal{M}, Q, q_0, F, \delta, \lambda_\phi, \Psi_\phi \rangle$, where

- $\lambda_\phi := \lambda \circ \phi$ (i.e. $\lambda_\phi(q,\sigma) := \phi(\lambda(q,\sigma))$),
- $\Psi\lambda_\phi := \Psi \circ \phi$ (i.e. $\Psi_\phi(q) := \phi(\Psi(q))$),

is said to be *obtained by mapping $\mathcal{T}$ with ϕ.*

The following proposition can be considerd as a variant of Theorem 2.1.22.

Proposition 5.6.2 *Let $\mathcal{T}_\phi$ be obtained from the subsequential transducer $\mathcal{T}$ by mapping with ϕ, where $\phi : \Sigma'^* \rightarrow \mathcal{M}$ is a homomorphism between the monoids Σ'^* and $\mathcal{M}$. Then $O_{\mathcal{T}_\phi} = O_{\mathcal{T}} \circ \phi$.*

Proposition 5.6.3 *Let $\Sigma' = \{1\}$ and $\mathcal{N} = \langle \mathbb{N}, +, 0 \rangle$. Then the function $\varphi : \Sigma'^* \rightarrow \mathbb{N}$ defined as*

$$\varphi(1^n) := n, \text{ for any } n \in \mathbb{N}$$

is a monoidal isomorphism between the monoids Σ'^* and $\mathcal{N}$.

The following properties are obvious:

- for any $a, b \in \Sigma'^*$, such that $|a| \leq |b|$ we have $\varphi(a^{-1}b) = \varphi(b) - \varphi(a)$,
- for any $a, b \in \Sigma'^*$, we have $\varphi(a \wedge b) = \min(\varphi(a), \varphi(b))$,
- for any $a \in \Sigma'^*$, we have $|a| = \varphi(a)$.

Proposition 5.6.4 *Let $\mathcal{T}$ be a classical subsequential transducer and let $\Sigma' = \{1\}$. Then $\mathcal{T}$ can be obtained by mapping $\mathcal{T}_\varphi$ with φ^{-1} i.e. $\mathcal{T} = \mathcal{T}_{\varphi\varphi^{-1}}$.*

The above proposition shows that mapping with φ realizes an isomorphism between classical subsequential transducers over output alphabets with one symbol and monoidal subsequential transducers over the monoid $\mathcal{N} = \langle \mathbb{N}, +, 0 \rangle$.

Corollary 5.6.5 *All results for transducer determinization (Section 5.2), deciding functionality (Section 4.6), deciding bounded variation (Section 5.3), and transducer minimization (Section 5.4) obtained for classical subsequential transducers are transferred directly[2] to monoidal subsequential transducers over the monoid $\mathcal{N}$.*

Remark 5.6.6 All results for transducer determinization, deciding functionality, deciding bounded variation, and transducer minimization obtained for classical subsequential transducers can be derived also for monoidal subsequential transducers over the monoid $\mathcal{R}^+ = \langle \mathbb{R}^+, +, 0 \rangle$ [Mohri, 2004] and other monoids [Gerdjikov and Mihov, 2017b].

5.7 Summing up

In this chapter we looked at deterministic finite-state devices for translating input strings. We first looked at deterministic monoidal finite-state transducers. (Recall that in our terminology monoidal finite-state transducers in general are non-deterministic). Monoidal *subsequential* transducers generalize deterministic monoidal finite-state transducers, as final states may produce their own output when reached at the end of the input. We then looked at determinization and minimization procedures.

Determinization procedures. We have shown that not all regular string functions can be recognized by means of classical subsequential transducers. A determinization procedure has been described which transforms a functional classical transducer with the bounded variation property into an equivalent classical subsequential transducer. We proof that a regular string function $f : \Sigma^* \to \Delta^*$ can be represented by a classical subsequential transducer iff f has the bounded variation property.

Bounded variation decision procedure. Afterwards we presented an effective bounded variation decision procedure and show how to integrate it in the determinization procedure for constructing classical subsequential transducer.

[2] Transferring the results the notions of a longest common prefix, remainder suffix and word length of words respectively have to be replaced by minimum, subtraction and value of natural numbers.

Minimization procedures. At the end of the chapter we presented minimization techniques for classical subsequential transducers. As a key step, the Myhill-Nerode relation for languages (cf. Def. 3.4.8) has been generalized to string functions (Def. 5.4.1).

5.8 Bibliographic notes

In our definition of a deterministic finite-state transducer (Definition 5.1.1), determinism refers to the transition behaviour and final states do not have output. In the literature, transducers with a deterministic transition behaviour without state output are sometimes called "sequential" transducers [Berstel, 1979]. Also Roches and Schabes [Roche and Schabes, 1997b] define sequential transducers in this way, but also demand that all states are final. This is motivated by applications in text rewriting, a situation where any string should be accepted as input. *Sub*sequentiality of transducers (Definition 5.1.4) points to the property that also final states may produce output ([Schützenberger, 1977]). The decidability of the bounded variation property is due to Choffrut [Choffrut, 1977]. The minimization algorithm for subsequential transducers was introduced by Mohri [Mohri, 2000]. Generalizations of the deterministic transducer constructions for other output monoids are presented in [Mohri, 2004, Gerdjikov and Mihov, 2017b, Gerdjikov, 2018].

5.9 Exercises for Chapter 5

Exercise 5.9.1 Let $\Sigma := \{a, b\}$. For each of the following string functions $f : \Sigma^* \to \Sigma^*$, describe a subsequential transducer, or - if possible - a deterministic transducer representing f:

1. $f(w) := wa$ if $|w|$ is odd, $f(w) = w$ otherwise,
2. the function that maps each word of the form wa to wb and conversely each word of the form wb to wa ($w \in \Sigma$).
3. the function that rewrites in each string w each occurrence of the infix ab in w to ba.
4. the function that rewrites in each string w each occurrence of the infix ab to ba with one possible exception: if ab is a suffix of w, this suffix is left untouched.

Exercise 5.9.2 Let A be a set. A function $d : A \times A \to \mathbb{R}$ is a distance on A if the following conditions hold:

1. $\forall a, b \in A : d(a, b) \geq 0$,
2. $\forall a, b \in A : d(a, b) = 0 \leftrightarrow a = b$,
3. $\forall a, b \in A : d(a, b) = d(b, a)$,
4. $\forall a, b, c \in A : d(a, b) + d(b, c) \geq d(a, c)$.

Verify that $d_S(u, v)$ defined in Definition 5.1.12 is a distance on Σ^*.

Exercise 5.9.3 Let Σ be a finite alphabet. Prove that every string function $f : \Sigma^* \to \Sigma^*$ with a finite range (i.e. $|codom(f)| < \infty$) can be represented by a classical subsequential transducer.

Exercise 5.9.4 Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the alphabet of decimal digits and $k \in \mathbb{N}^+$ be an arbitrary number. Describe the equivalence classes of the Myhill-Nerode relation for the function $f : \Sigma^* \to \Sigma^*$, defined as:

1. $f(a) = b$ if the number represented by b is equal to the whole number division of the number represented by a by k and b has no leading zeros;
2. $f(a) = b$ if the number represented by b is equal to the multiplication of the number represented by a by k and b has no leading zeros.

Exercise 5.9.5 Modify the definition of $R_{\mathcal{T}}$ in Proposition 5.4.6 in order to preserve its validity in the case that the transition function δ is a partial function.

Exercise 5.9.6 Construct the minimal subsequential finite-state transducer for the first function from Exercise 5.9.4 for $k = 3$ by:

1. constructing the corresponding Myhill-Nerode transducer for f;
2. starting from the 2-tape constructed by the corresponding algorithm given in Example 8.4.4 and applying the determinization, canonization and minimization procedures.

Chapter 6

Bimachines

In the previous chapter we have seen that not all regular string functions can be represented by deterministic or subsequential transducers. In this chapter we look at a more powerful concept. We introduce bimachines, a deterministic finite-state device that exactly represents the class of all regular string functions. Bimachines have been introduced in [Schützenberger, 1961] and have been treated also in [Eilenberg, 1974, Berstel, 1979, Roche and Schabes, 1997b].

6.1 Basic definitions

Following our earlier procedure we generalize the classical concept of a bimachine to the more general situation where output values are in a monoid.

Definition 6.1.1 A *monoidal bimachine* is a tuple $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ where

- $\mathcal{M} = \langle M, \circ, e \rangle$ is a monoid,
- $\mathcal{A}_L = \langle \Sigma, L, s_L, L, \delta_L \rangle$ and $\mathcal{A}_R = \langle \Sigma, R, s_R, R, \delta_R \rangle$ are deterministic finite-state automata called the *left* and *right automaton* of the bimachine;
- $\psi : (L \times \Sigma \times R) \to M$ is a partial function called the *output function*.

Note that all states of $\mathcal{A}_L$ and $\mathcal{A}_R$ are final. If $\mathcal{M}$ is a free monoid, then $\mathcal{B}$ is called a *classical bimachine*. In this case we simply assume that the output alphabet coincides with the alphabets of the left and right automata, Σ, and write $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$.

Definition 6.1.2 Let $\mathcal{M} = \langle M, \circ, e \rangle$ be a monoid, let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ denote a monoidal bimachine, let Σ denote the alphabet of $\mathcal{A}_L$ and $\mathcal{A}_R$. Consider an input sequence $t = \sigma_1 \sigma_2 \cdots \sigma_n \in \Sigma^*$ $(n \geq 0)$ with letters σ_i $(i = 1, \ldots, n)$. If all the values $\delta_L^*(\sigma_1 \sigma_2 \cdots \sigma_i)$ and $\delta_R^*(\sigma_n \sigma_{n-1} \cdots \sigma_i)$ are defined $(1 \leq i \leq n)$ we obtain a pair of paths

$$\begin{array}{lccccccc} \pi_L : & l_0 & \to^{\sigma_1} & l_1 & \to \ldots & l_{i-1} & \to^{\sigma_i} & l_i \quad \to \ldots \quad l_{n-1} \to^{\sigma_n} l_n \\ \pi_R : & r_0 & \leftarrow^{\sigma_1} & r_1 & \leftarrow \ldots & r_{i-1} & \leftarrow^{\sigma_i} & r_i \quad \leftarrow \ldots \quad r_{n-1} \leftarrow^{\sigma_n} r_n \end{array}$$

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

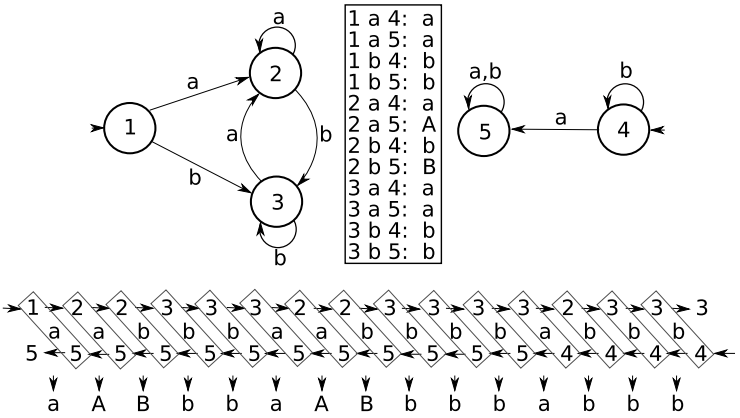
1 a 4: a
1 a 5: a
1 b 4: b
1 b 5: b
2 a 4: a
2 a 5: A
2 b 4: b
2 b 5: B
3 a 4: a
3 a 5: a
3 b 4: b
3 b 5: b
a
b
a,b
1 2 2 3 3 3 2 2 3 3 3 3 2 3 3 3
a a b b b a a b b b b a b b b
5 5 5 5 5 5 5 5 5 5 5 5 4 4 4 4
a A B b b a A B b b b a b b b

Figure 6.1: Example for a classical bimachine, cf. Example 6.1.3. The bimachine translates letters a and b into uppercase letters if the left neighbour is a and another a follows later in the text. We see a pair of successful paths for input *aabbbaabbbbabbb*. Boxes show the triples used for computing the output components.

where π_L is a path of the left automaton $\mathcal{A}_L$ starting from $l_0 := s_L$ and π_R is a path of the right automaton $\mathcal{A}_L$ (arrows indicate reading order) starting from $r_n := s_R$. If all outputs $\psi(l_{i-1}, \sigma_i, r_i)$ are defined ($1 \leq i \leq n$), then we call (π_L, π_R) a *pair of successful paths* of $\mathcal{B}$ with label $\sigma_1\sigma_2 \ldots \sigma_n$ and output

$$O_{\mathcal{B}}(t) := \psi(l_0, \sigma_1, r_1) \circ \psi(l_1, \sigma_2, r_2) \circ \ldots \circ \psi(l_{i-1}, \sigma_i, r_i) \circ \ldots \circ \psi(l_{n-1}, \sigma_n, r_n).$$

In the special case where $t = \varepsilon$ we have $O_{\mathcal{B}}(t) = e$. The partial function $O_{\mathcal{B}}$ is called the *output function* of the bimachine, or the *function represented by the bimachine*. If $O_{\mathcal{B}}(t) = m$ we say that the bimachine $\mathcal{B}$ *translates* t *into* m.

The output corresponds to the product of the outputs defined by parallel steps in the two paths in the following way:

$$\begin{array}{lccccc} \pi_L: & l_0 \to^{\sigma_1} l_1 & \to \ldots & l_{i-1} \to^{\sigma_i} l_i & \to \ldots & l_{n-1} \to^{\sigma_n} l_n \\ \pi_R: & r_0 \leftarrow^{\sigma_1} r_1 & \leftarrow \ldots & r_{i-1} \leftarrow^{\sigma_i} r_i & \leftarrow \ldots & r_{n-1} \leftarrow^{\sigma_n} r_n \\ & \psi(l_0, \sigma_1, r_1) & \ldots & \psi(l_{i-1}, \sigma_i, r_i) & \ldots & \psi(l_{n-1}, \sigma_n, r_n) \end{array}$$

From the definition we see immediately that bimachines enable a computation *linear* in the length of the input: to compute the output for an input string t we may first read t in the reverse order and memorize the sequence of states of π_R ("back"). Afterwards we read t in the natural order ("forth"). With each transition $l_{i-1} \to^{\sigma_i} l_i$ in the left automaton we may directly produce the output component $\psi(l_{i-1}, \sigma_i, r_i)$. As a matter of fact, instead of using a "back-and-forth" method we can also use a "forth-and-back" strategy.

Example 6.1.3 Figure 6.1 shows a classical bimachine, the left (right) automaton has three (two) states. Since in a bimachine each state is final we do not mark final states. In the figure we see a pair of successful paths with output *aABbbaABbbbabbb* for the input string *aabbbaabbbbabbb*. The states of the left

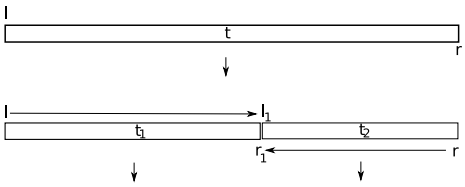

Figure 6.2: Illustration for the principle described in Proposition 6.1.5. The output of a string t under the generalized output function for input $t = t_1t_2$ starting from states l, r can split into outputs for t_1 and t_2 using the additional states $l_1 = \delta_L^*(l, t_1)$ and $r_1 = \delta_R^*(r, \rho(t_2))$.

automaton reached after reading each input letter are shown above the input word. The right automaton reads in the converse direction, states reached are shown under the input word. Note that at a certain point of the input text the right automaton is in state 5 iff a symbol a follows somewhere on the right. The output for a particular occurrence of an input symbol σ is determined by σ, the upper left automaton state reached before σ, and the lower right automaton state after σ.

Similarly as in the case of monoidal finite-state automata, also for bimachines the path-based view is not the only way to define the behaviour of the machine. In the case of finite-state automata we have seen that the *generalized* transition function (cf. Definition 2.1.14) can be used to directly define the language of an automaton. A parallel concept in the bimachine case is the following.

Definition 6.1.4 Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ denote a monoidal bimachine. The *generalized output function* ψ^* of $\mathcal{B}$ is inductively defined as follows:

- $\psi^*(l, \varepsilon, r) = e$ for all $l \in L, r \in R$;
- $\psi^*(l, t\sigma, r) = \psi^*(l, t, \delta_R(r, \sigma)) \circ \psi(\delta_L^*(l, t), \sigma, r)$ for $l \in L, r \in R, t \in \Sigma^*, \sigma \in \Sigma$.

In the definition we use our general convention that whenever we write an expression where a partial function is applied to an argument we always assume that the value is defined. This means that ψ^* is a partial function. We want to show that the partial mapping

$$\Sigma^* \to M : t \mapsto \psi^*(s_L, t, s_R)$$

coincides with the output function of the bimachine. As a preparation the following proposition shows how output values for a string $t = t_1 \cdot t_2 \in \Sigma^*$ can be traced back to outputs for the substrings t_1 and t_2. Here we assume that the traversals in the left and right automaton start at arbitrary states. See Figure 6.2 for an illustration.

Proposition 6.1.5 *Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ be a monoidal bimachine and $\mathcal{M} = \langle M, \circ, e \rangle$. Let $t = t_1 \cdot t_2 \in \Sigma^*$. Then for all states $l \in L$ and $r \in R$ we have*

$$\psi^*(l, t, r) = \psi^*(l, t_1, \delta_R^*(r, \rho(t_2))) \circ \psi^*(\delta_L^*(l, t_1), t_2, r).$$

Proof. The proof is by induction on $|t_2|$. For $|t_2| = 0$ we have $t = t_1$, $t_2 = \varepsilon$ and $\delta_R^*(r, \rho(t_2)) = r$. The definition of ψ^* implies that

$$\begin{aligned}\psi^*(l,t,r) &= \psi^*(l,t,r) \circ e \\ &= \psi^*(l,t_1,r) \circ \psi^*(\delta_L^*(l,t_1),\varepsilon,r) \\ &= \psi^*(l,t_1,\delta_R^*(r,\rho(t_2))) \circ \psi^*(\delta_L^*(l,t_1),t_2,r).\end{aligned}$$

For the induction step let $t_2 = t_3\sigma$. Using the induction hypothesis for t_1 and t_3 and the definition of ψ^* we obtain

$$\begin{aligned}& \psi^*(l,t,r) \\ = \;& \psi^*(l,t_1t_3\sigma,r) \\ = \;& \psi^*(l,t_1t_3,\delta_R(r,\sigma)) \circ \psi(\delta_L^*(l,t_1t_3),\sigma,r) \\ = \;& \psi^*(l,t_1,\delta_R^*(\delta_R(r,\sigma),\rho(t_3))) \circ \psi^*(\delta_L^*(l,t_1),t_3,\delta_R(r,\sigma)) \circ \psi(\delta_L^*(l,t_1t_3),\sigma,r) \\ = \;& \psi^*(l,t_1,\delta_R^*(r,\rho(t_3\sigma))) \circ \psi^*(\delta_L^*(l,t_1),t_3\sigma,r) \\ = \;& \psi^*(l,t_1,\delta_R^*(r,\rho(t_2))) \circ \psi^*(\delta_L^*(l,t_1),t_2,r). \quad \square\end{aligned}$$

Using Proposition 6.1.5 is now simple to see that the partial mapping $\Sigma^* \to M : t \mapsto \psi^*(s_L, t, s_R)$ in fact coincides with the output function of the bimachine. Given a text t we only need to apply the proposition in an iterative manner until the text is split into single-letter subtexts. We then arrive at the description of the output via pairs of successful paths. Yet another more "local" way of looking at bimachines can be derived from the following notion.

Definition 6.1.6 Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ be a monoidal bimachine. An *edge* of $\mathcal{B}$ is a sixtupel $E = \langle l, r, \sigma, m, l', r' \rangle$ such that $\sigma \in \Sigma$, l, l' are states of $\mathcal{A}_L$, r, r' are states of $\mathcal{A}_R$, $\delta_L(l, \sigma) = l')$, $\delta_R(r', \sigma) = r$, and $\psi(l, \sigma, r') = m$. The *left state pair* of E is $\langle l, r \rangle$, the *right state pair* of E is $\langle l', r' \rangle$.

Edges are written in the form

$$\begin{array}{c} l \to^{\sigma} l' \\ r \leftarrow^{\sigma} r' \\ m \end{array}$$

A sequence of edges $E_1, \ldots, E_n$ is called *admissible* if the right state pair of E_i coincides with the left state pair of E_{i+1} for $i = 1, \ldots, n-1$. Admissible sequences of edges are also called *chains*. A chain $E_1, \ldots, E_n$ is called *complete* if the left state pair of E_1 contains the start state s_L of $\mathcal{A}_L$ and the right state pair of E_n contains the start state s_R of $\mathcal{A}_R$. Clearly, each pair of successful paths

$$\begin{array}{ccccccc} l_0 \to^{\sigma_1} l_1 & \to \ldots & l_{i-1} \to^{\sigma_i} l_i & \to \ldots & l_{n-1} \to^{\sigma_n} l_n \\ r_0 \leftarrow^{\sigma_1} r_1 & \leftarrow \ldots & r_{i-1} \leftarrow^{\sigma_i} r_i & \leftarrow \ldots & r_{n-1} \leftarrow^{\sigma_n} r_n \\ \psi(l_0, \sigma_1, r_1) & \ldots & \psi(l_{i-1}, \sigma_i, r_i) & \ldots & \psi(l_{n-1}, \sigma_n, r_n) \end{array}$$

corresponds to a unique complete chain

$$\begin{array}{ccccccc} l_0 \to^{\sigma_1} l_1 & \ldots & l_{i-1} \to^{\sigma_i} l_i & \ldots & l_{n-1} \to^{\sigma_n} l_n \\ r_0 \leftarrow^{\sigma_1} r_1 & \ldots & r_{i-1} \leftarrow^{\sigma_i} r_i & \ldots & r_{n-1} \leftarrow^{\sigma_n} r_n \\ \psi(l_0, \sigma_1, r_1) & \ldots & \psi(l_{i-1}, \sigma_i, r_i) & \ldots & \psi(l_{n-1}, \sigma_n, r_n) \end{array}$$

and vice versa.

Remark 6.1.7 Most often, bimachines are used as devices for text rewriting. In this context the above concept is too general and bimachines should be restricted in the sense that the left and right automaton, the generalized output function ψ^* and the function represented by the bimachine $O_{\mathcal{B}}$ are *total*.

Definition 6.1.8 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. A monoidal bimachine $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi\rangle$ is *total* iff the transition functions of $\mathcal{A}_L$ and $\mathcal{A}_R$ are total and if the output function $\psi : (L \times \Sigma \times R) \to M$ is total.

Remark 6.1.9 Clearly, for every bimachine $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi\rangle$ we always have $O_{\mathcal{B}}(\varepsilon) = e$. This deficiency can be eliminated by amending the bimachine with additional parameters for specifying the desired mapping of ε and a flag for specifying whether $O_{\mathcal{B}}(\varepsilon)$ is defined. We will further consider bimachines without such amendments but all results can be extended in this obvious way to obtain a special treatment of the empty word input.

Relationship between deterministic transducers and bimachines

Remark 6.1.10 Every *deterministic monoidal finite-state transducer* mapping the empty word to e can be regarded as a monoidal bimachine with left automaton coinciding with the underlying automaton of the transducer and trivial right automaton consisting of *one state* with a loop transition for each symbol. Since there is only one state in the right automaton the bimachine output function coincides with the transducer output function λ.

Every *monoidal subsequential transducer* mapping the empty word to e can be represented by a monoidal bimachine with left automaton coinciding with the underlying automaton of the transducer and a right automaton consisting of *two states* p and q: the initial state p has a transition for each symbol to q, and the non-initial state q has a loop transition for each symbol. The bimachine output function coincides with the transducer output function λ if the right state is q. Otherwise the output is the concatenation of the transducer transition output λ with the Ψ output of the transducer destination state. As an illustration, consider the classical subsequential transducer in the upper part of Figure 6.3 and the corresponding bimachine and its behaviour for input strings a and ab shown below.

Example 6.1.11 In Remark 5.1.11 we argued that the regular function $R = \langle a, a\rangle^* \cup (\langle a, \varepsilon\rangle^* \cdot \langle b, b\rangle)$ cannot be represented by a subsequential transducer. Consider the classical bimachine $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi\rangle$ shown in Figure 6.4. For the text $t = aab$ we obtain the following pair of successful paths.

$$\begin{array}{ccc} l \to^a & l \to^a & l \to^b l' \\ r'' \leftarrow^a r'' & \leftarrow^a r'' & \leftarrow^b r \\ \varepsilon = \psi(l, a, r'') & \varepsilon = \psi(l, a, r'') & b = \psi(l, b, r) \end{array}$$

When reading a letter a, from the back path and the state r'' the bimachine knows that a final letter b will follow and hence can produce the correct output ε.

As a matter of fact, this example also shows that not all regular functions represented by bimachines can be represented by means of deterministic or subsequential transducers.

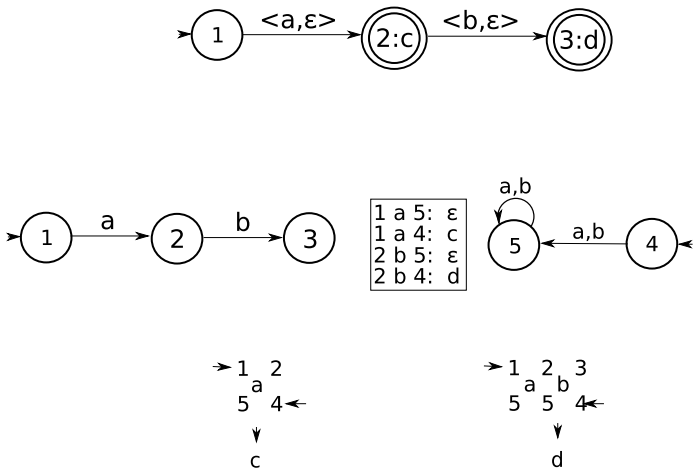

Figure 6.3: Representation of subsequential transducers via bimachines. The classical subsequential transducer in the upper part represents the function $\{\langle a, c\rangle, \langle ab, d\rangle\}$. The corresponding bimachine is shown below.

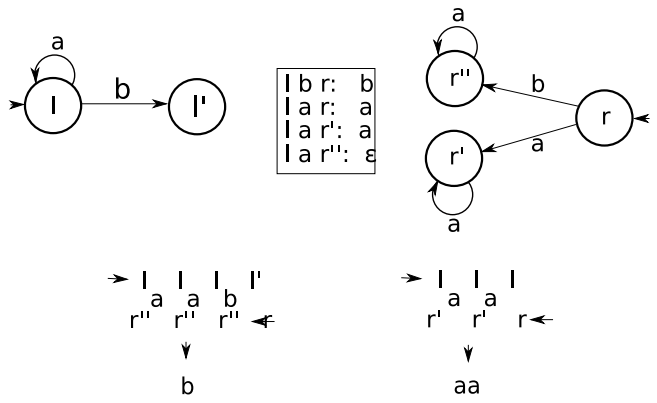

Figure 6.4: Bimachine representing the regular function $\langle a, a\rangle^* \cup (\langle a, \varepsilon\rangle^* \cdot \langle b, b\rangle)$. The function cannot be represented by a subsequential transducer.

6.2 Equivalence of regular string functions and classical bimachines

In this section we prove the correspondence between regular string functions and classical bimachines. The result is well-known and goes back to [Schützenberger, 1961]. Our first aim is to show that classical bimachines only accept regular string functions.

From bimachines to regular string functions. The central step for this direction can be generalized to the monoidal case. Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ be a monoidal bimachine. A pair of successful paths

$$\begin{array}{ccccccc} l_0 \to^{\sigma_1} l_1 & \to \ldots & l_{i-1} \to^{\sigma_i} l_i & \to \ldots & l_{n-1} \to^{\sigma_n} l_n \\ r_0 \leftarrow^{\sigma_1} r_1 & \leftarrow \ldots & r_{i-1} \leftarrow^{\sigma_i} r_i & \leftarrow \ldots & r_{n-1} \leftarrow^{\sigma_n} r_n \\ m_1 & \ldots & m_i & \ldots & m_n \end{array}$$

for $t = \sigma_1 \ldots \sigma_n$ can be represented as a path in the Cartesian product $L \times R$ of the form

$$\langle l_0, r_0 \rangle \to^{\sigma_1}_{m_1} \langle l_1, r_1 \rangle \to \ldots \langle l_{i-1}, r_{i-1} \rangle \to^{\sigma_i}_{m_i} \langle l_i, r_i \rangle \to \ldots \langle l_{n-1}, r_{n-1} \rangle \to^{\sigma_n}_{m_n} \langle l_n, r_n \rangle$$

This perspective leads us to the construction of a monoidal finite-state transducer equivalent to the bimachine.

Proposition 6.2.1 *For each monoidal bimachine $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ there exists a monoidal finite-state transducer $\mathcal{A} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$, such that $O_{\mathcal{B}} = L(\mathcal{A})$.*

Proof. Consider the real-time transducer

$$\mathcal{A} := \langle \Sigma^* \times \mathcal{M}, L \times R, \{s_L\} \times R, L \times \{s_R\}, \Delta \rangle$$

where Δ contains all transitions $\langle l, r \rangle \to^{\sigma}_{m} \langle l', r' \rangle$ such that $\langle l, r, \sigma, m, l', r' \rangle$ is an edge of $\mathcal{B}$. Each successful path in the transducer $\mathcal{A}$ corresponds to unique complete chain of the bimachine $\mathcal{B}$. The aforementioned correspondence between complete chains and pairs of successful paths shows that the output function of the bimachine and the transducer language coincide. □

As a matter of fact, in general the transducers obtained from the above translation of bimachines are non-deterministic. This follows from the fact that there exist regular functions that can be represented by bimachines, but not by means of any deterministic or subsequential transducer, as we have seen in Example 6.1.11.

Example 6.2.2 Figure 6.5 shows the bimachine $\mathcal{B}$ introduced in Example 6.1.11, which has two edges of the form $\langle l, r', a, a, l, r \rangle$ and $\langle l, r', a, a, l, r' \rangle$ (among other edges). Below we see the transducer resulting from the translation. It has the two transitions $\langle l, r' \rangle \to^{a}_{a} \langle l, r \rangle$ and $\langle l, r' \rangle \to^{a}_{a} \langle l, r' \rangle$ (among other transitions) and is non-deterministic.

Corollary 6.2.3 *For each bimachine $\mathcal{B}$ the function represented by $\mathcal{B}$ is a regular string function.*

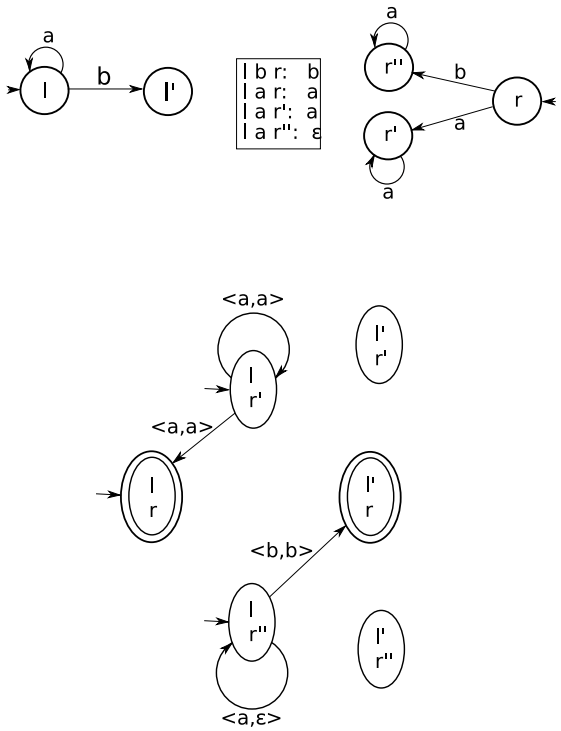

Figure 6.5: A bimachine and its translation into a transducer. The resulting transducer is non-deterministic.

Proof. This follows directly from the Proposition 6.2.1 (classical case) and Theorem 4.5.5 since bimachines have a functional input-output behaviour. □

Remark 6.2.4 An important property of the transducer constructed in Proposition 6.2.1 is that for each pair $\langle u, m\rangle \in L(\mathcal{A})$ there exists exactly one successful path in $\mathcal{A}$. Transducers with this property are called *unambiguous*.

From regular string functions to bimachines. We now show that conversely each regular string function can be represented as a bimachine. We follow the approach presented in [Gerdjikov et al., 2017]. For convenience we restrict attention to string functions f such that $f(\varepsilon) = \varepsilon$ if $f(\varepsilon)$ is defined. As our starting point we use the result that each regular string function can be represented as a classical real-time finite-state transducer (cf. Theorem 4.5.5 and Proposition 4.4.8). It remains to be shown that classical real-time finite-state transducers can be translated into bimachines. This step can be described in the more general monoidal setting.

Proposition 6.2.5 *Let $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta\rangle$ be a monoidal trimmed functional real-time transducer with output in the monoid $\mathcal{M} = \langle M, \circ, e\rangle$ such that $\langle \varepsilon, e\rangle \in L(\mathcal{T})$. Then there exists a monoidal bimachine $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi\rangle$ such that $L(\mathcal{T}) = O_{\mathcal{B}}$.*

Proof. The underlying finite-state automaton (cf. Definition 4.4.3) of $\mathcal{T}$ does not have ε-transitions. We may apply the determinization procedure described in Remark 3.2.3.

The *right deterministic automaton* of the bimachine $\mathcal{A}_R = \langle \Sigma, Q_R, s_R, Q_R, \delta_R\rangle$ is defined as the result when applying the determinization procedure to the reversed underlying automaton of $\mathcal{T}$ and setting all states to final. This means that $Q_R \subseteq 2^Q$, $s_R = F$ and

$$\delta_R(R, a) := \{q \in Q \mid \exists q' \in R, m \in M : \langle q, \langle a, m\rangle, q'\rangle \in \Delta\}.$$

A *state selector function* is a partial function $\phi : Q_R \to Q$ selecting for a non-empty set $P \in Q_R$ an element $p = \phi(P) \in P$. A state of the *left deterministic automaton* $\mathcal{A}_L = \langle \Sigma, Q_L, s_L, Q_L, \delta_L\rangle$ is a pair consisting of a subset of Q and state selector function ϕ. This implies that $Q_L \subseteq 2^Q \times 2^{Q_R \times Q}$ and therefore Q_L is finite. The following induction defines the states and the transition function of the left automaton:

- $s_L := \langle I, \phi_0\rangle$ where $\phi_0(R) := \begin{cases} \text{any element of } R \cap I & \text{if } R \cap I \neq \emptyset \\ \text{undefined} & \text{otherwise.} \end{cases}$
- For $\langle L, \phi\rangle \in Q_L$ and $a \in \Sigma$ we define $\delta_L(\langle L, \phi\rangle, a) := \langle L', \phi'\rangle$ where
  - $L' := \{q' \mid \exists q \in L, m \in M : \langle q, \langle a, m\rangle, q'\rangle \in \Delta\}$.
  - $\phi'(R') := \begin{cases} \text{any element of } \{q' \in R' \mid \exists m \in M : \langle q, \langle a, m\rangle, q'\rangle \in \Delta\} \\ \quad \text{if } q = \phi(\delta_R(R', a)) \text{ is defined} \\ \text{undefined} \quad \text{otherwise.} \end{cases}$

The definition of the transition function δ_R implies that if $q = \phi(\delta_R(R', a))$ is defined, then the set $\{q' \in R' \mid \exists m \in M : \langle q, \langle a, m\rangle, q'\rangle \in \Delta\}$ is non-empty and $\phi'(R')$ can be defined in the above way.

It remains to define the *output function* ψ of the bimachine. Given a pair of states $\langle L, \phi\rangle$ and R' of the left and right automaton and $a \in \Sigma$, let $\langle L', \phi'\rangle := \delta_L(\langle L, \phi\rangle, a)$ and $R := \delta_R(R', a)$. Then

$$\psi(\langle L, \phi\rangle, a, R') := \begin{cases} \text{any element of} \\ \{m \mid \langle \phi(R), \langle a, m\rangle, \phi'(R')\rangle \in \Delta\} & \text{if } \phi(R) \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

If $q = \phi(R)$ is defined, then $\{q' \in R' \mid \exists m \in M : \langle q, \langle a, m\rangle, q'\rangle \in \Delta\}$ is non-empty. The definition of $\phi'(R')$ implies that $\{m \mid \langle \phi(R), \langle a, m\rangle, \phi'(R')\rangle \in \Delta\}$ is non-empty and $\psi(\langle L, \phi\rangle, a, R')$ can be defined in the above way.

We show that the function defined by the bimachine $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi\rangle$ coincides with the language of the transducer $\mathcal{T}$. Let $u \in dom(L(\mathcal{T}))$ where $u = a_1 \dots a_k$. Consider a successful path with label $\langle a_1 \dots a_k, m\rangle$ of $\mathcal{T}$:

$$\pi = q_0 \quad \to^{\langle a_1, m_1\rangle} \quad q_1 \quad \to^{\langle a_2, m_2\rangle} \quad q_2 \quad \to^{\langle a_3, m_3\rangle} \quad \dots \quad \to^{\langle a_k, m_k\rangle} \quad q_k.$$

Claim 1: for any $i \in \{0, \dots, k-1\}$, if $\langle L_i, \phi_i\rangle = \delta_L^*(s_L, a_1 \dots a_i)$ and $R_i = \delta_R^*(s_R, a_k a_{k-1} \dots a_{i+1})$, then $\phi_i(R_i)$ is defined.

The claim is proved by induction. For $i = 0$, since π is a successful path we have $I \cap \delta_R^*(s_R, a_k a_{k-1} \dots a_1) \neq \emptyset$ and therefore $\phi_0(R_0)$ is defined. For the induction step assume that $\phi_i(R_i)$ is defined. Then since $R_i = \delta_R(R_{i+1}, a_{i+1})$ according the definition of δ_L we have that $\phi_{i+1}(R_{i+1})$ is defined.

Claim 2: the path π'

$$\pi' = q'_0 \quad \to^{\langle a_1, m'_1\rangle} \quad q'_1 \quad \to^{\langle a_2, m'_2\rangle} \quad q'_2 \quad \to^{\langle a_3, m'_3\rangle} \quad \dots \quad \to^{\langle a_k, m'_k\rangle} \quad q'_k,$$

where $q'_i = \phi_i(R_i)$ and $m'_i = \psi(\langle L_i, \phi_i\rangle, a_{i+1}, R_{i+1})$ is a successful path in $\mathcal{T}$.

We prove Claim 2. Our definitions imply that for any $\langle L, \phi\rangle \in Q_L$, $a \in \Sigma$ and $R' \in Q_R$ such that $\phi(\delta(R', a))$ is defined we have

$$\langle \phi(\delta(R', a)), \langle a, \psi(\phi, a, R')\rangle, \phi'(R')\rangle \in \Delta,$$

where $\langle L', \phi'\rangle = \delta_L(\langle L, \phi\rangle)$. Therefore π' is a path of $\mathcal{T}$. By definition $q_0 = \phi_0(R_0) \in I$ and $q_k = \phi_k(R_k) \in R_k = F$. Hence π' is a successful path in $\mathcal{T}$.

It thus follows that $\langle a_1 \dots a_k, m'_1 m'_2 \dots m'_k\rangle \in L(\mathcal{T})$. Since $\mathcal{T}$ is functional we have $m'_1 m'_2 \dots m'_k = O_{\mathcal{B}}(u) = m$. Summing up, we have seen that $u \in dom(L(\mathcal{T}))$ implies $u \in dom(L(\mathcal{B}))$ and $L(\mathcal{T})(u) = L(\mathcal{B})(u)$.

On the other hand, if $u \notin dom(L(\mathcal{T}))$, then $I \cap R_0 = I \cap \delta_R^*(s_R, a_k a_{k-1} \dots a_1)$ is empty and $\phi_0(I \cap R_0)$ is undefined. It follows that $u \notin dom(L(\mathcal{B}))$. Hence $L(\mathcal{T})$ and $L(\mathcal{B})$ have the same domain and coincide. □

Corollary 6.2.6 *For each regular string function f that maps the empty word to ε there exists a bimachine $\mathcal{B}$ representing f.*

Proof. This follows directly from Theorem 4.5.5 and Proposition 6.2.5. □

Combining Corollary 6.2.3 and Corollary 6.2.6 we derive the following theorem.

Theorem 6.2.7 *The class of regular string functions that map the empty word to ε coincides with the class of functions represented by bimachines.*

Remark 6.2.8 From Corollary 6.2.6 and Remark 6.2.4 it follows that for every regular string function f that maps the empty word to ε there exists an unambiguous transducer recognizing f.

The construction presented in Proposition 6.2.5 does not require that the input transducer is functional. In fact it can be applied to any real-time transducer.

Proposition 6.2.9 *Let $\mathcal{T} = \langle \Sigma^* \times \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal trimmed real-time transducer with output in the monoid $\mathcal{M} = \langle M, \circ, e \rangle$ such that $\langle \varepsilon, e \rangle \in L(\mathcal{T})$. Then there exists a monoidal bimachine $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ such that $O_{\mathcal{B}} \subseteq L(\mathcal{T})$ and $dom(O_{\mathcal{B}}) = L_{\times 2}(\mathcal{T})$.*

Proof. We construct the monoidal bimachine $\mathcal{B}$ by applying the construction from the proof of Proposition 6.2.5. Claim 1 from the proof remains valid and therefore $dom(O_{\mathcal{B}}) = L_{\times 2}(\mathcal{T})$. Considering Claim 2 the path π' built is a path in $\mathcal{T}$ and therefore $O_{\mathcal{B}} \subseteq L(\mathcal{T})$. □

6.3 Pseudo-minimization of monoidal bimachines

A monoidal bimachine consists of two deterministic finite-state automata and an output function. When we want to minimize a monoidal bimachine, we cannot minimize the two component automata in a naïve way: since all states of the left and right automaton are final the classical minimization procedure would lead to automata with a single state only. In this way we would not obtain an equivalent bimachine. This shows that the minimization procedure has to take into account the bimachine output function as well.

Definition 6.3.1 Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ be a monoidal bimachine with component automata $\mathcal{A}_L = \langle \Sigma, L, s_L, L, \delta_L \rangle$ and $\mathcal{A}_R = \langle \Sigma, R, s_R, R, \delta_R \rangle$. The *left profile function* $\psi_L : L \to 2^{\Sigma \times R \times \Sigma^*}$ is defined as

$$\psi_L(q_L) := \{ \langle \sigma, q_R, \psi(q_L, \sigma, q_R) \rangle \mid \sigma \in \Sigma, q_R \in R \}.$$

The *set of left profiles* of $\mathcal{B}$ is defined as $\Gamma_L := \{ \psi_L(q_L) \mid q_L \in L \}$.

For a given state $q_L \in L$ the set $\psi_L(q_L)$ is called the *state profile for* q_L. It gives a complete view on the way how q_L contributes to the output. For each symbol $\sigma \in \Sigma$ and state q_R of the right automaton $\mathcal{A}_R$ the state profile specifies the output $\psi(q_L, \sigma, q_R)$. Given the left profile of each state q_L we may derive the bimachine output function: $\psi(q_L, \sigma, q_R)$ is the unique monoid element m such that $\langle \sigma, q_R, m \rangle \in \psi_L(q_L)$.

For minimization we cannot simply identify two states q_L and q'_L with the same state profile - we need to know that with the same input sequences we always reach from q_L and q'_L two states that again have the same state profile. However, we may think of $\psi_L(q_L)$ as a colour of q_L and consider the coloured

automaton $\mathcal{A}_L^c = \langle \Sigma, \Gamma_L, L, s_L, L, \delta_L, \psi_L \rangle$, the set of all state profiles representing the set of colours. The minimization procedure for coloured deterministic finite-state automata described in the second part of Section 3.6 leads to an equivalent minimal coloured deterministic finite-state automaton

$$\mathcal{A}_L^{c\,\prime} = \langle \Sigma, \Gamma_L, \{[q]_R \mid q \in L\}, [s_L]_R, \{[q]_R \mid q \in L\}, \delta', col' \rangle$$

where $col'([q]_R) := col(q)$ (see Corollary 3.6.16). We may use

$$\mathcal{A}_L{}' := \langle \Sigma, \Gamma_L, \{[q]_R \mid q \in L\}, [s_L]_R, \{[q]_R \mid q \in L\}, \delta' \rangle$$

instead of $\mathcal{A}_L$ as the left automaton and the colouring col' to define the new output function. The equation $col'([q]_R) := col(q)$ directly ensures that the new bimachine has the same output function as $\mathcal{B}$.

In the general case of coloured deterministic finite-state automata, the computation of classes of equivalent states is based on the relations R_i $(i \geq 0)$, which were defined in the following way:

$$\begin{aligned} q\ R_0\ p &\leftrightarrow (q \in F \leftrightarrow p \in F)\ \&\ (q \in F \rightarrow col(q) = col(p)) \\ q\ R_{i+1}\ p &\leftrightarrow q\ R_i\ p\ \&\ \forall a \in \Sigma : \delta(q,a)\ R_i\ \delta(p,a). \end{aligned}$$

In the situation considered here, all states are final. Hence the definition can be simplified:

$$\begin{aligned} q\ R_0\ p &\leftrightarrow \psi_L(q) = \psi_L(p) \\ q\ R_{i+1}\ p &\leftrightarrow q\ R_i\ p\ \&\ \forall a \in \Sigma : \delta(q,a)\ R_i\ \delta(p,a). \end{aligned}$$

Corollary 6.3.2 *The equivalence relation S for minimizing the left automaton $\mathcal{A}_L$ taking bimachine output into account coincides with the relation $R = \bigcap_0^\infty R_i$ where*

1. $R_0 = ker_L(\psi_L)$
2. $R_{i+1} = \bigcap_{a \in \Sigma} ker_L(f_a^{(i)}) \cap R_i$

For $a \in \Sigma$ and $i \in \mathbb{N}$ the function $f_a^{(i)} : L \rightarrow L/R_i$ is defined as

$$f_a^{(i)}(q) := [\delta_L(q,a)]_{R_i}.$$

As mentioned above the right automaton $\mathcal{A}_R$ of the bimachine can be minimized in exactly the dual way.

Definition 6.3.3 Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ be a monoidal bimachine. Then the monoidal bimachine $\mathcal{B}' = \langle \mathcal{M}, \mathcal{A}'_L, \mathcal{A}'_R, \psi' \rangle$ constructed by the minimization of $\mathcal{A}_L$ and $\mathcal{A}_R$ (which are treated as coloured automata as described above) is called the *pseudo-minimal monoidal bimachine equivalent to* $\mathcal{B}$.

Remark 6.3.4 In general the pseudo-minimal bimachine equivalent to a given classical bimachine $\mathcal{B}$ is not minimal. Moving parts of the outputs along paths may lead to a smaller pseudo-minimal bimachine [Reutenauer and Schützenberger, 1991].

6.4 Direct composition of classical bimachines

Clearly, classical bimachines over the same alphabet Σ are closed under composition. This follows from the closure properties of classical 2-tape automata (Proposition 4.3.5) and the equivalence of functional 2-tape automata and bimachines presented in the previous section. From an application point of view, composition of classical bimachines is extremely useful when trying to achieve efficient solutions for iterated or cascaded forms of text translation. Hence the question arises how bimachines can be *directly* composed in a simple way. In this section we present a construction that builds the resulting bimachine for the composition directly from the source bimachines.

Idea

Let two bimachines $\mathcal{B}' = \langle \mathcal{A}'_L, \mathcal{A}'_R, \psi' \rangle$ and $\mathcal{B}'' = \langle \mathcal{A}''_L, \mathcal{A}''_R, \psi'' \rangle$ over the same alphabet Σ be given, with left and right deterministic automata of the form $\mathcal{A}'_L = \langle \Sigma, L', s'_L, L', \delta'_L \rangle$, $\mathcal{A}'_R = \langle \Sigma, R', s'_R, R', \delta'_R \rangle$ and $\mathcal{A}''_L = \langle \Sigma, L'', s''_L, L'', \delta''_L \rangle$, $\mathcal{A}''_R = \langle \Sigma, R'', s''_R, R'', \delta''_R \rangle$. The composition of the two output functions over an input $t = a_1 a_2 \dots a_n, a_i \in \Sigma, i = 1, 2, \dots, n$ is illustrated below:

$$
\begin{array}{ccccccccccc}
l'_0 & \to^{a_1} & l'_1 & \to \dots & l'_{i-1} & \to^{a_i} & l'_i & \to \dots & \to^{a_n} & l'_n \\
r'_0 & \leftarrow^{a_1} & r'_1 & \leftarrow \dots & r'_{i-1} & \leftarrow^{a_i} & r'_i & \leftarrow \dots & \leftarrow^{a_n} & r'_n \\
 & v_1 & & \dots & & v_i & & \dots & v_n & \\
l''_0 & \to^{v_1} & l''_1 & \to \dots & l''_{i-1} & \to^{v_i} & l''_i & \to \dots & \to^{v_n} & l''_n \\
r''_0 & \leftarrow^{\rho(v_1)} & r''_1 & \leftarrow \dots & r''_{i-1} & \leftarrow^{\rho(v_i)} & r''_i & \leftarrow \dots & \leftarrow^{\rho(v_n)} & r''_n \\
 & w_1 & & \dots & & w_i & & \dots & w_n &
\end{array}
$$

where $l'_0 = s'_L$, $r'_n = s'_R$, $l''_0 = s''_L$, $r''_n = s''_R$, $v_i = \psi'(l'_{i-1}, a_i, r'_k)$, $w_i = \psi''^*(l''_{i-1}, v_i, r''_k)$ for $i = 1, 2, \dots, n$. The upper successful pair of paths of the first bimachine is used to translate the input sequence $a_1 \dots a_n$ into the string $v_1 \cdot \dots \cdot v_n$. The lower successful pair of paths of the second bimachine - depicted in a condensed representation - translates the latter string into $w_1 \cdot \dots \cdot w_n$. We have $O_{\mathcal{B}''}(O_{\mathcal{B}'}(t)) = w_1 \cdot w_2 \cdot \dots \cdot w_n$. In order to compose the two pairs of paths into one we consider the following scheme:

$$
\begin{array}{ccccccccc}
\langle l'_0, r'_0, l''_0 \rangle & \to^{a_1} & \dots & \langle l'_{i-1}, r'_{i-1}, l''_{i-1} \rangle & \to^{a_i} & \langle l'_i, r'_i, l''_i \rangle & \dots & \to^{a_n} & \langle l'_n, r'_n, l''_n \rangle \\
\langle l'_0, r'_0, r''_0 \rangle & \leftarrow^{a_1} & \dots & \langle l'_{i-1}, r'_{i-1}, r''_{i-1} \rangle & \leftarrow^{a_i} & \langle l'_i, r'_i, r''_i \rangle & \dots & \leftarrow^{a_n} & \langle l'_n, r'_n, r''_n \rangle \\
 & w_1 & \dots & & & w_i & & \dots & w_n
\end{array}
$$

where $l'_0 = s'_L$, $r'_n = s'_R$, $l''_0 = s''_L$, $r''_n = s''_R$, $w_i = \psi''^*(l''_{i-1}, \psi'(l'_{i-1}, a_i, r'_i), r''_i)$ for $i = 1, 2, \dots, n$. This simplified picture suggests that the states of the *left* (resp. *right*) automaton of the composed bimachine are triples encoding

- two states l', r' respectively reached by the left and right automaton of the first bimachine, plus
- a state l'' (r'') of the *left* (resp. *right*) automaton of the second bimachine.

We shall see that the information encoded in the new states is rich enough to simulate the composition of the two given bimachines. However, in the form presented above, the left and right automaton of the combined machine in general

are non-deterministic. Thus, to obtain a bimachine as a result of the composition, the two product automata have to be replaced by deterministic variants. We arrive at the following formal construction.

Formal construction

Let $\mathcal{B}' = \langle \mathcal{A}'_L, \mathcal{A}'_R, \psi' \rangle$ and $\mathcal{B}'' = \langle \mathcal{A}''_L, \mathcal{A}''_R, \psi'' \rangle$ be two bimachines over the same alphabet Σ, let $\mathcal{A}'_L$, $\mathcal{A}'_R$, $\mathcal{A}''_L$, and $\mathcal{A}''_R$ as above. Let

$$\begin{aligned} \mathcal{A}^N_L &:= \langle \Sigma, L' \times R' \times L'', \{s'_L\} \times R' \times \{s''_L\}, L' \times R' \times L'', \Delta_L \rangle \\ \mathcal{A}^N_R &:= \langle \Sigma, L' \times R' \times R'', L' \times \{s'_R\} \times \{s''_R\}, L' \times R' \times R'', \Delta_R \rangle \end{aligned}$$

where

1. Δ_L contains all transitions of the form $\langle \langle l'_1, r'_1, l''_1 \rangle, a, \langle l'_2, r'_2, l''_2 \rangle \rangle$ such that $\delta'_L(l'_1, a) = l'_2$, $\delta'_R(r'_2, a) = r'_1$, $\delta''^*_L(l''_1, \psi'(l'_1, a, r'_2)) = l''_2$,
2. Δ_R contains all transitions of the form $\langle \langle l'_2, r'_2, r''_2 \rangle, a, \langle l'_1, r'_1, r''_1 \rangle \rangle$ such that $\delta'_L(l'_1, a) = l'_2$, $\delta'_R(r'_2, a) = r'_1$, $\delta''^*_R(r''_2, \rho(\psi'(l'_1, a, r'_2))) = r''_1$.

Let $\mathcal{A}_L$ and $\mathcal{A}_R$ be the deterministic finite state automata constructed from $\mathcal{A}^N_L$ and $\mathcal{A}^N_R$ by the subset construction given in Theorem 3.2.2, where the states are restricted to the ones reachable from the starting states. I.e.

$$\begin{aligned} \mathcal{A}_L &= \langle \Sigma, Q_L, s_L, Q_L, \delta_L \rangle \\ \mathcal{A}_R &= \langle \Sigma, Q_R, s_R, Q_R, \delta_R \rangle \end{aligned}$$

where

$$\begin{aligned} s_L &= \{s'_L\} \times R' \times \{s''_L\} \\ s_R &= L' \times \{s'_R\} \times \{s''_R\} \\ \hat{\delta}_L(A, a) &= \{\langle l'_2, r'_2, l''_2 \rangle \mid \exists \langle l'_1, r'_1, l''_1 \rangle \in A : \langle \langle l'_1, r'_1, l''_1 \rangle, a, \langle l'_2, r'_2, l''_2 \rangle \rangle \in \Delta_L\} \\ \hat{\delta}_R(A, a) &= \{\langle l'_1, r'_1, r''_1 \rangle \mid \exists \langle l'_2, r'_2, r''_2 \rangle \in A : \langle \langle l'_2, r'_2, r''_2 \rangle, a, \langle l'_1, r'_1, r''_1 \rangle \rangle \in \Delta_R\} \\ Q_L &= \{A \in 2^{L' \times R' \times L''} \mid \exists v \in \Sigma^* : A = \hat{\delta}^*_L(s_L, v)\} \\ Q_R &= \{A \in 2^{L' \times R' \times R''} \mid \exists v \in \Sigma^* : A = \hat{\delta}^*_R(s_R, v)\} \\ \delta_L &= \hat{\delta}_L|_{Q_L \times \Sigma} \\ \delta_R &= \hat{\delta}_R|_{Q_R \times \Sigma} \end{aligned}$$

For states A and B of $\mathcal{A}_L$ and $\mathcal{A}_R$ and $a \in \Sigma$ define

$$\psi(A, a, B) := w \text{ iff}$$

there exist tuples $\langle l'_1, r'_1, l''_1 \rangle \in A$ and $\langle l'_2, r'_2, r''_2 \rangle \in B$ such that there exist $l''_2 \in L''$ and $r''_1 \in R''$ such that

$$\begin{aligned} \langle \langle l'_1, r'_1, l''_1 \rangle, a, \langle l'_2, r'_2, l''_2 \rangle \rangle &\in \Delta_L, \\ \langle \langle l'_2, r'_2, r''_2 \rangle, a, \langle l'_1, r'_1, r''_1 \rangle \rangle &\in \Delta_R, \end{aligned}$$

and $w = \psi''^*(l''_1, \psi'(l'_1, a, r'_2), r''_2)$.

Proposition 6.4.1 *Let $\mathcal{B}'$, $\mathcal{B}''$ be two bimachines over the same alphabet Σ as above, let the bimachine $\mathcal{B}$ and its output function ψ be defined as above. Then ψ is well-defined and $\mathcal{B} := \langle \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ represents the composition of the bimachines $\mathcal{B}'$ and $\mathcal{B}''$, i.e. $O_{\mathcal{B}} = O_{\mathcal{B}'} \circ O_{\mathcal{B}''}$.*

Proof. We first prove the following auxiliary proposition:

For every state $A \in Q_L$ we have:

$$\begin{aligned} \mid A_{\times 2 \times 3} \mid &= 1 \\ \forall r \in R' : \mid A \cap (L' \times \{r\} \times L'') \mid &\leq 1 \end{aligned}$$

The proof is by induction on the length of the shortest string which is the label of a path from s_L to A in $\mathcal{A}_L$. If the length is 0, then the state $A = s_L$ and the proposition follows trivially.
Let us assume that the proposition holds for all states in Q_L, reachable with less than n symbols. Let $v_2 = a_1 a_2 \dots a_n \in \Sigma^*$ be one of the shortest words which is a label of a path from s_L to A. Let the last transition on the path be $A' \rightarrow^{a_n} A$, i.e. $A = \delta_L(A', a)$. By the induction hypothesis the proposition holds for A'. Since $\mid A'_{\times 2 \times 3} \mid = 1$ there exists $l' \in L'$ such that $A'_{\times 2 \times 3} = \{l'\}$. From the definitions of δ_L and Δ_L it follows that $A_{\times 2 \times 3} = \{l\}$ for $l = \delta'_L(l', a_n)$ and hence $\mid A_{\times 2 \times 3} \mid = 1$. Let us now assume that there exists an $r \in R'$ such that $\mid A \cap (L' \times \{r'\} \times L'') \mid > 1$. Then there exist $k_1, k_2 \in L''$ such that $k_1 \neq k_2$ and $\langle l, r, k_1 \rangle, \langle l, r, k_2 \rangle \in A$. From the definition of Δ_L it follows that there exist $\langle l', r'_1, k'_1 \rangle, \langle l', r'_2, k'_2 \rangle \in A'$ such that $r'_1 = \delta'_R(r, a_n) = r'_2 =: r'$. But from the induction hypothesis for A' it follows that for $r' \in R'$ we have $\mid A' \cap (L' \times \{r'\} \times L'') \mid \leq 1$ and hence $k'_1 = k'_2$. The definition of Δ_L implies that $k_1 = \delta''_L(k'_1, \psi'(l', a_n, r)) = \delta''_L(k'_2, \psi'(l', a_n, r)) = k_2$, which is a contradiction. Hence $\forall r \in R' : \mid A \cap (L' \times \{r\} \times L'') \mid \leq 1$.

In the same way we prove the dual proposition:

For each state $B \in Q_R$ we have:

$$\begin{aligned} \mid B_{\times 1 \times 3} \mid &= 1 \\ \forall l \in L' : \mid B \cap (\{l\} \times R' \times R'') \mid &\leq 1 \end{aligned}$$

Now we are ready to prove that ψ is well-defined. Let $\psi(A, a, B) := w$. Then there exist tuples $\langle l'_1, r'_1, l''_1 \rangle \in A$ and $\langle l'_2, r'_2, r''_2 \rangle \in B$ and $l''_2 \in L'$ and $r''_1 \in R''$ such that

$$\begin{aligned} \langle \langle l'_1, r'_1, l''_1 \rangle, a, \langle l'_2, r'_2, l''_2 \rangle \rangle &\in \Delta_L, \\ \langle \langle l'_2, r'_2, r''_2 \rangle, a, \langle l'_1, r'_1, r''_1 \rangle \rangle &\in \Delta_R, \end{aligned}$$

and $w = \psi''^*(l''_1, \psi'(l'_1, a, r'_2), r''_2)$. From the auxiliary proposition for A it follows that l'_1 is uniquely defined and from the auxiliary proposition for B it follows that r'_2 is uniquely defined. But since $\langle \langle l'_1, r'_1, l''_1 \rangle, a, \langle l'_2, r'_2, l''_2 \rangle \rangle \in \Delta_L$ it follows that r'_1 and l'_2 are uniquely defined. In this case from the auxiliary proposition for A we know that l''_1 is uniquely defined and for B we know that r''_2 is uniquely defined. Hence $w = \psi''^*(l''_1, \psi'(l'_1, a, r'_2), r''_2)$ is uniquely defined. Furthermore,

we have seen that (†) for any given triple (A, a, B) such that $\psi(A, a, B) = w$ is defined, we have *unique* tuples $\langle l'_1, r'_1, l''_1\rangle \in A$ and $\langle l'_2, r'_2, r''_2\rangle \in B$ and $l''_2 \in L'$ and $r''_1 \in R''$ with the aforementioned properties.

Now we have to show that $O_{\mathcal{B}} = O_{\mathcal{B}'} \circ O_{\mathcal{B}''}$.

("$\subseteq$") Let $O_{\mathcal{B}}(t) = w$ and $t = a_1 \cdots a_n$. Then in $\mathcal{B}$ we have a pair of successful paths

$$\begin{array}{ccccccccccc} A_0 & \to^{a_1} & A_1 & \to \ldots & A_{i-1} & \to^{a_i} & A_i & \to \ldots & \to^{a_n} & A_n \\ B_0 & \leftarrow^{a_1} & B_1 & \leftarrow \ldots & B_{i-1} & \leftarrow^{a_i} & B_i & \leftarrow \ldots & \leftarrow^{a_n} & B_n \\ & w_1 & & \ldots & & w_i & & \ldots & w_n & \end{array}$$

such that $w = w_1 \cdots w_n$. The property † shows that from each triple (A_{i-1}, a_i, B_i) we obtain unique tuples $\left\langle l'_{i-1,1}, r'_{i-1,1}, l''_{i-1,1}\right\rangle \in A_i$ and $\left\langle l'_{i,2}, r'_{i,2}, r''_{i,2}\right\rangle \in B_i$ and $l''_{i,2} \in L'$ and $r''_{i-1,1} \in R''$ such that

$$\begin{array}{rcl} \left\langle \left\langle l'_{i-1,1}, r'_{i-1,1}, l''_{i-1,1}\right\rangle, a, \left\langle l'_{i,2}, r'_{i,2}, l''_{i,2}\right\rangle\right\rangle & \in & \Delta_L, \\ \left\langle \left\langle l'_{i,2}, r'_{i,2}, r''_{i,2}\right\rangle, a, \left\langle l'_{i-1,1}, r'_{i-1,1}, r''_{i-1,1}\right\rangle\right\rangle & \in & \Delta_R, \end{array}$$

and $w_i = \psi''^*(l''_{i-1,1}, \psi'(l'_{i-1,1}, a, r'_{i,2}), r''_{i,2})$. The definition of δ_L shows that $\left\langle l'_{i,2}, r'_{i,2}, l''_{i,2}\right\rangle \in A_i$. We also have $\left\langle l'_{i,1}, r'_{i,1}, l''_{i,1}\right\rangle \in A_i$. Now the first auxiliary proposition shows that $l'_{i,2} = l'_{i,1}$ for $i = 0, \ldots n-1$. The definition of δ_R shows $\left\langle l'_{i-1,1}, r'_{i-1,1}, r''_{i-1,1}\right\rangle \in B_{i-1}$. We also have $\left\langle l'_{i-1,2}, r'_{i-1,2}, r''_{i-1,2}\right\rangle \in B_{i-1}$. The second auxiliary proposition shows that $r'_{i-1,1} = r'_{i-1,2}$ for $i = 1, \ldots n$. Using the second statements shown in the auxiliary propositions we see that always

$$\begin{array}{rcl} \left\langle l'_{i,2}, r'_{i,2}, l''_{i,2}\right\rangle = \left\langle l'_{i,1}, r'_{i,1}, l''_{i,1}\right\rangle & =: & \left\langle l'_i, r'_i, l''_i\right\rangle \\ \left\langle l'_{i-1,1}, r'_{i-1,1}, r''_{i-1,1}\right\rangle = \left\langle l'_{i-1,2}, r'_{i-1,2}, r''_{i-1,2}\right\rangle & =: & \left\langle l'_{i-1}, r'_{i-1}, r''_{i-1}\right\rangle. \end{array}$$

In the nondeterministic automata $\mathcal{A}^N_L$ and $\mathcal{A}^N_R$ we thus obtain a pair of paths

$$\begin{array}{ccccccccc} \langle l'_0, r'_0, l''_0\rangle & \to^{a_1} & \ldots & \left\langle l'_{i-1}, r'_{i-1}, l''_{i-1}\right\rangle & \to^{a_i} & \langle l'_i, r'_i, l''_i\rangle & \ldots & \to^{a_n} & \langle l'_n, r'_n, l''_n\rangle \\ \langle l'_0, r'_0, r''_0\rangle & \leftarrow^{a_1} & \ldots & \left\langle l'_{i-1}, r'_{i-1}, r''_{i-1}\right\rangle & \leftarrow^{a_i} & \langle l'_i, r'_i, r''_i\rangle & \ldots & \leftarrow^{a_n} & \langle l'_n, r'_n, r''_n\rangle \\ & w_1 & \ldots & & w_i & & \ldots & w_n & \end{array}$$

such that $w_i = \psi''^*(l''_{i-1}, \psi'(l'_{i-1}, a, r'_i), r''_i)$ for $i = 1, \ldots n$. Obviously, from this pair we can reconstruct two pairs of successful paths in $\mathcal{B}'$ and $\mathcal{B}''$ respectively leading to output $w = w_1 \cdots w_n$ in the composition of $\mathcal{B}'$ and $\mathcal{B}''$.

("$\supseteq$") It is simple to see that conversely each two pairs of successful paths in $\mathcal{B}'$ and $\mathcal{B}''$ where the first pair of paths in $\mathcal{B}'$ provides the input for the second pair of paths in $\mathcal{B}''$ correspond to a pair of paths in the nondeterministic automata $\mathcal{A}^N_L$ and $\mathcal{A}^N_R$, which in turn corresponds to a pair of successful paths in the bimachine $\mathcal{B}$. □

6.5 Summing up

In this chapter we looked at monoidal bimachines, which deterministically translate input strings into elements of a monoid. The most important subcase is the situation where the monoid is free (classical bimachines).

We have given distinct formal descriptions of the output function of a bimachine und showed how to represent deterministic transducers as bimachines. We mentioned the important result that classical bimachines are equivalent to arbitrary regular string functions. A pseudo-minimization technique for monoidal bimachines was presented. We also introduced a direct method for composing classical bimachines over the same alphabet.

6.6 Exercises for Chapter 6

Exercise 6.6.1 Describe a procedure for deciding the equivalence of classical bimachines.

Exercise 6.6.2 Let $\mathcal{T}$ be a trimmed unambiguous functional monoidal real-time transducer. Let $\mathcal{B}$ be the monoidal bimachine constructed from $\mathcal{T}$ by the procedure from Proposition 6.2.5. Let $\mathcal{T}'$ be the trimmed unambiguous functional monoidal real-time transducer constructed from $\mathcal{B}$ by the procedure from Proposition 6.2.1. Show that $\mathcal{T}'$ is isomorphic to $\mathcal{T}$.

Exercise 6.6.3 Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ be a monoidal bimachine. Let the monoidal bimachine $\mathcal{B}' = \langle \mathcal{M}, \mathcal{A}'_L, \mathcal{A}'_R, \psi' \rangle$ be the pseudo-minimal monoidal bimachine equivalent to $\mathcal{B}$. Let the monoidal bimachine $\mathcal{B}'' = \langle \mathcal{M}, \mathcal{A}''_L, \mathcal{A}''_R, \psi' \rangle$ be the monoidal bimachine obtained after applying the pseudo-minimisation procedure from Section 6.3 to $\mathcal{B}'$. Prove that $\mathcal{B}''$ is isomorphic to $\mathcal{B}'$.

Exercise 6.6.4 Calculate an asymptotic upper bound for the complexity of the composition of two bimachines.

Part II

From theory to practice

After having introduced the central mathematical notions and constructions in the field of finite-state automata, transducers, and bimachines, we now turn to practice. We first introduce the programming language *C(M)* used in this book. Afterwards *C(M)*-implementations for all algorithms and constructions introduced in the previous chapters are given. In the final Chapters 9 - 11 the techniques for building and implementing automata are further illustrated by looking at some concrete automaton constructions arising in distinct applications fields. The aim here is not a comprehensive survey of important applications. Rather, the constructions described at the end of this part are meant to give a first impression of the richness of applications where finite-state technology offers efficient solutions. In many cases, programs presented could be further optimized using more complex code.

Chapter 7

The *C(M)* language

In this chapter we introduce the *C(M)* language, a new programming language created by the first co-author of this book. The language will be used throughout the rest of the book for implementing and presenting algorithms. *C(M)* statements and expressions closely resemble the notation commonly used for the presentation of formal constructions in a Tarskian style set theoretical language. The usual set theoretic objects such as sets, functions, relations, tuples etc. are naturally integrated in the language. In contrast to imperative languages such as C or Java [Pratt and Zelkowitz, 2000], *C(M)* is a functional declarative programming language. *C(M)* has many similarities with Haskell [Hutton, 2007] but makes use of the standard mathematical notation like SETL [Schwartz et al., 1986]. When implementing the solution to a problem, instead of specifying how to achieve it, we specify the goal itself. In practice, we just formally describe the kind of mathematical object we want to obtain. This allows us to focus on the high-level mathematical steps of a construction as opposed to the low-level implementation details. The *C(M)* compiler translates a well-formed *C(M)* program into efficient C code, which can be executed after compilation. Since it is easy to read *C(M)* programs, a pseudo-code description becomes obsolete. The programs presented below are tested and fully functional and can be compiled without modifications and run on a computer. The compiler for the *C(M)* language is freely available at `http://lml.bas.bg/~stoyan/lmd/C(M).html`.

In the first section we start with a simple algorithm showing the flavour of the language. Afterwards we provide a more comprehensive overview of the language elements.

7.1 Basics and simple examples

Perhaps the best starting point for an introduction to *C(M)* is a short selection of simple examples.

Example 7.1.1 Recall that the composition of two binary relations R_1 and R_2 is defined as $R_1 \circ R_2 := \{\langle a, c\rangle \mid \exists b : \langle a, b\rangle \in R_1, \langle b, c\rangle \in R_2\}$ (Def. 1.1.3). In *C(M)*, the usual elements for describing sets are available - we may define this

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

composition directly as

$$\text{compose}(R_1, R_2) := \{(a, c) \mid (a, b) \in R_1, (b, c) \in R_2\};$$

Given the two relations R_1 and R_2, the above function returns the set of pairs (a, c) where (a, b) runs over R_1 (b is arbitrary) and (b, c) runs over R_2. Mathematically, the *n-th order composition* $R^{\langle n \rangle}$ of a binary relation R is defined in an inductive manner:

- $R^1 := R$,
- $R^{i+1} := R^i \circ R$

Explicit inductive constructions are a core element of *C(M)*. These definitions start with a base step such as "step 1" where we define a base form of the objects to be constructed. Then "step i+1" explains how to obtain variant $i + 1$ of the objects from variant i. An "until" clause explains when the construction is finished. Following this scheme, the n-th order composition may be introduced in *C(M)* as the object "$\text{compose}_\text{n}(R, n)$" in the following way:

$\text{compose}_\text{n}(R, n) := R'$, **where**
  $R' :=$ **induction**
    step 1 :
      $R'^{(1)} := R$;
    step $i + 1$:
      $R'^{(i+1)} := \text{compose}(R'^{(i)}, R)$;
    until $i = n$
    ;
  ;

It is shown below how to write this pretty-print version of the code. Mathematically, the transitive closure of a binary relation R is defined (see Definition 1.1.8) as the relation

$$C_R := \bigcup_{i=1}^{\infty} R^i.$$

To construct C_R we may proceed inductively, defining $C_R^{(n)} = \bigcup_{i=1}^{n} R^i$. For a finite relation the inductive step has to be performed until $R^{\langle n+1 \rangle} \subseteq C_R^{(n)}$. In *C(M)* we may use exactly the same procedure and construct C_R using the following induction:

$C_R :=$ **induction**
  step 1 :
    ${C_R}^{(1)} := R$;
  step $n + 1$:
    ${C_R}^{(n+1)} := {C_R}^{(n)} \cup \text{compose}_\text{n}(R, n + 1)$;
  until $\text{compose}_\text{n}(R, n + 1) \subseteq {C_R}^{(n)}$
  ;

Program 7.1.2 In order to complete this *C(M)* program we have to specify the type of each object. The resulting program has the following form:

```
1  RECL is 2^(IN×IN);
2  compose : RECL × RECL → RECL;
3  compose(R1, R2) := {(a, c) | (a, b) ∈ R1, (b, c) ∈ R2};
4  compose_n : RECL × IN → RECL;
5  compose_n(R, n) := R', where
6      R' := induction
7          step 1 :
8              R'^(1) := R;
9          step i + 1 :
10             R'^(i+1) := compose(R'^(i), R);
11         until i = n
12         ;
13     ;
14 transitiveClosure : RECL → RECL;
15 transitiveClosure(R) := C_R, where
16     C_R := induction
17         step 1 :
18             C_R^(1) := R;
19         step n + 1 :
20             C_R^(n+1) := C_R^(n) ∪ compose_n(R, n + 1);
21         until compose_n(R, n + 1) ⊆ C_R^(n)
22         ;
23     ;
```

In Line 1 we define the type $\mathcal{RECL}$ as the sets of pairs of natural numbers. Line 2 defines the type of "compose" as a function that takes two relations and returns a relation. Line 3 is the actual definition of the function "compose". Line 4 defines the type of the N-th order composition "compose$_N$" as a function that takes a relation and a natural number and returns a relation. Lines 5-13 present the definition of the function "compose$_N$". Line 14 defines the type of "transitiveClosure" as a function from relations to relations. Lines 15-23 present the inductive definition of the function "transitiveClosure".

How to write, compile and use *C(M)* programs

From a graphical point of view, the above *C(M)* programs are presented using bookstyle mathematical notion and formatted using page style. As a matter of fact, when writing real programs one has to specify the description in plain text first. Yet, *C(M)* comes with a nice feature: after writing a *C(M)* program as plain text, the compiler can translate it into LaTeX for producing the above layout. The plain text description for Program 7.1.2 looks as follows:

```
REL is 2^(IN*IN);
```

```
compose in REL * REL -> REL;
compose(R_1,R_2) := {(a,c) | (a,b) in R_1, (b,c) in R_2};

compose_n in REL * IN -> REL;
compose_n(R,n) := R', where
   R' := induction
      step 1 :
         R'@1 := R;
      step i+1 :
         R'@i+1 := compose(R'@i,R);
      until i=n
      ;
   ;

transitiveClosure in REL -> REL;
transitiveClosure(R) := C_R, where
   C_R := induction
      step 1:
         C_R@1 := R;
      step n+1:
         C_R@n+1 := C_R@n \/ compose_n(R,n+1);
      until compose_N(R,n+1) subseteq C_R@n
      ;
   ;
```

The *C(M)* compiler takes as input the plain text of the program and outputs a C source code (or a LaTeX code). Let the above program be stored as a file named `Program_6.1.2.cm`. Then the compiler is invoked by

```
cm Program_6.1.2.cm -o Program_6.1.2.c
```

The file `Program_6.1.2.c` will contain the corresponding C source code. Afterwards the C code has to be compiled with the `gcc` compiler for producing an executable. Currently the C code generated by *C(M)* contains nested functions, which are supported by `gcc` but are not ANSI C compatible. The compilation is invoked by the following command

```
gcc -fnested-functions -o Program_6.1.2 Program_6.1.2.c
```

In the newer versions (e.g 4.7) of `gcc` the option `-fnested-functions` has to be omitted. The LaTeX layout is generated by the compiler in the following way:

```
cm -L Program_6.1.2.cm -o Program_6.1.2.tex
```

Afterwards the Program_6.1.2.tex file has to be compiled with LaTeX.

The way HOW we describe objects in programs matters

In mathematics, a fixed object can be described in many distinct ways. Ignoring matters of transparency, it is not important how we specify the object. However,

in a computational context the way how we describe an object may have a strong influence on the time needed to compute it. Under this perspective, the above construction of the transitive closure and Program 7.1.2 can easily be improved. Although this program produces the correct transitive closure when specifying a concrete finite relation R, it has two sources of inefficiency. First, the definition of the composition in Line 3 runs through all pairs in R_1, all pairs in R_2, and then it checks whether the selected pairs are of the form (a, b) and (b, c). This can be optimized by explicitly constructing a function F_R mapping a given b to the set $\{c \,|\, (b, c) \in R_2\}$. We can then define the composition by running through the pairs (a, b) in R_1 and the elements c in $F_R(b)$. Second, in Line 20 of the above construction we construct for each n the $n+1$-st order composition. As a side effect we repeat the computation of the n-th, $n-1$-th, ..., 2-nd order compositions, which have been already computed in previous steps. Moreover, if we have a pair, (a, b), which is in R^{i_1}, R^{i_2}, ..., then the extensions $\{(a, c) \,|\, \exists b : (b, c) \in R\}$ will be in R^{i_1+1}, R^{i_2+1}, ... Since it is not relevant in which step (a, b) is generated we can optimize our construction by generating the extensions of (a, b) only once.

Program 7.1.3 The following improved construction avoids the two deficiencies. First, it explicitly builds the function F_R, and the composition is performed in the optimized way. And second, the induction is performed by considering at each step one new pair from the set C_R. In this way, starting from ${C_R}^{(0)} := R$, in step $n+1$ we generate the composition of the $n+1$-st element in $C_R^{(n)}$ with the relation R until the set $C_R^{(n)}$ is exhausted.

The program below presents two new features of *C(M)*, the use of a subcase analysis in definitions (Line 9) and the use of the "functionalize"-operator $\mathcal{F}$ in Line 3, cf. Definition 1.1.16.

```
1  transitiveClosure : 2^(IN×IN) → 2^(IN×IN);
2  transitiveClosure(R) := C_R, where
3      F_R := F_{1→2}(R);
4      C_R := induction
5          step 0 :
6              C_R^(0) := R;
7          step n+1 :
8              (a, b) := (C_R^(n) as (IN × IN)*)_{n+1};
9              C_R^(n+1) := { C_R^(n) ∪ {(a, c) | c ∈ F_R(b)}   if !F_R(b)
                            { C_R^(n)                           otherwise;
10         until n = |C_R^(n)|
11         ;
12     ;
```

The plain text version of the above program has the following form:

```
transitiveClosure in 2^(IN*IN) -> 2^(IN*IN);
transitiveClosure(R) := C_R, where
```

```
F_R := Func(1,2,R);
C_R := induction
   step 0:
      C_R@0 := R;
   step n+1:
      (a,b) := (C_R@n as (IN*IN)^*)[n+1];
      C_R@n+1 := ? C_R@n \/  {(a,c) | c in F_R(b)} if !F_R(b)
                 ? C_R@n otherwise;
   until n=|C_R@n|
   ;
;
```

In Line 3 the function F_R is defined. The closure C_R of R is defined in the induction starting from Line 4. In Line 6 the base value $C_R^{(0)}$ is defined to be R. In step n+1 of the construction, the pair (a, b) is defined as the $n+1$-st element of $C_R^{(n)}$ (Line 8). The expression $(C_R^{(n)}$**as** $(\mathbb{N} \times \mathbb{N})^*)$ means that we treat the set of pairs $C_R^{(n)}$ as a list of pairs. In this case the order of the elements in the list is the order in which the elements have been added to the set. In Line 9 we define the value $C_R^{(n+1)}$ using a subcase analysis. If $F_R(b)$ is defined, we join $C_R^{(n)}$ with the composition of (a, b) with R. Otherwise, the composition is empty and we define $C_R^{(n+1)} = C_R^{(n)}$. The induction ends when the condition in Line 10 is met, i.e., when the set $C_R^{(n)}$ is exhausted.

C(M) program for sorting a list of numbers

Program 7.1.4 As another illustrative example, the following program implements the merge sort algorithm for sorting a list of natural numbers in increasing order with respect to a (partial) order relation R. We first define a function "merge" that takes

- two sorted lists L_1 and L_2 of natural numbers, and
- a partial ordering represented as a function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$.

The function merges the two lists into a single sorted list L using an induction. During the induction, a natural number k_1 is maintained in parallel. This number represents the position of the first candidate of L_1 to be added to the merged result list (all earlier elements of the first list have been added at this point to the final list already). Using k_1 and the length of the current preliminary final list the number of elements of the second list that have been added to the final list already is found. In this way we obtain the position k_2 (Line 8) of the second list where we find the next candidate $(L_2)_{k_2}$ of L_2 to be integrated into the final list. At the beginning at induction step 0, the merged result list is empty and the preliminary value for k_1 is 1 (Line 6). At induction step $i + 1$ either $(L_2)_{k_2}$ or $(L_1)_{k_1}$ (i.e., the k_1-th element of L_1) are added as the next element to the final list. To this end the numbers $(L_1)_{k_1}$ and $(L_2)_{k_2}$

are compared with respect to the input relation R. The value for k_1 is updated accordingly.

$$
\begin{array}{ll}
1 & \text{merge} : \mathbb{N}^* \times \mathbb{N}^* \times (\mathbb{N} \times \mathbb{N} \to \mathbb{B}) \to \mathbb{N}^*; \\
2 & \text{merge}(L_1, L_2, R) := L, \ \textbf{where} \\
3 & \quad (L, k_1) \in \mathbb{N}^* \times \mathbb{N}; \\
4 & \quad (L, k_1) := \textbf{induction} \\
5 & \qquad \textbf{step } 0 : \\
6 & \qquad\quad (L^{(0)}, {k_1}^{(0)}) := (\varepsilon, 1); \\
7 & \qquad \textbf{step } i+1 : \\
8 & \qquad\quad k_2 := \left|L^{(i)}\right| - {k_1}^{(i)} + 2; \\
9 & \qquad\quad (L^{(i+1)}, {k_1}^{(i+1)}) := \\
 & \qquad\quad \begin{cases} (L^{(i)} \cdot \langle (L_2)_{k_2} \rangle, {k_1}^{(i)}) & \textbf{if } ({k_1}^{(i)} > |L_1|) \\ & \quad \vee ((k_2 \le |L_2|) \wedge \mathrm{R}((L_2)_{k_2}, (L_1)_{{k_1}^{(i)}})) \\ (L^{(i)} \cdot \langle (L_1)_{{k_1}^{(i)}} \rangle, {k_1}^{(i)} + 1) & \textbf{otherwise;} \end{cases} \\
10 & \qquad \textbf{until } ({k_1}^{(i)} > |L_1|) \wedge (\left|L^{(i)}\right| - {k_1}^{(i)} + 2 > |L_2|) \\
11 & \qquad ; \\
12 & \quad ; \\
13 & \text{sort} : \mathbb{N}^* \times (\mathbb{N} \times \mathbb{N} \to \mathbb{B}) \to \mathbb{N}^*; \\
14 & \text{sort}(L, R) := \begin{cases} L & \\ \quad \textbf{if } |L| \le 1 & \\ \text{merge}(\text{sort}((L)_{1,\ldots,|L|/2}, R), \text{sort}((L)_{|L|/2+1,\ldots,|L|}, R), R) & \\ \quad \textbf{otherwise;} & \end{cases}
\end{array}
$$

If the input list L has at most one element, then the input list itself is returned in Line 14. Otherwise L is split into two sublists in the middle. The two parts are first sorted and then merged using the function merge.

7.2 Types, terms, and statements in *C(M)*

We will now give a more thorough description of the language elements. In this section we describe types, terms and statements.

Types

C(M) is a strongly typed language. The type of every object has to be either explicitly stated or implicitly derived from its definition. There is no universal type. The types in *C(M)* are inductively defined. *Basic types* in *C(M)* are:

- $\mathbb{N}$ natural number,
- $\mathbb{Z}$ integer number,
- $\mathbb{R}$ real number, and
- $\mathbb{B}$ Boolean values.

In plain text, these types are respectively written `IN`, `IZ`, `IR`, `IB`. *C(M)* supports the following ways to build *composite types*:

- Tuples: if $T_1, T_2, \ldots, T_n$ are types, then $T_1 \times T_2 \times \ldots \times T_n$ is the type of n-tuples where the i-th projection is of type T_i.

- Lists: if T is any type, then T^* is the type of lists with elements of type T.
- Sets: if T is any type, then 2^T is the type of sets with elements of type T.
- Functions: if T_1 and T_2 are types, then $T_1 \to T_2$ is the type of functions with domain T_1 and range T_2.

In plain text, these types are respectively written `T_1 * T_2 * ... * T_n`, `T^*`, `2^T`, `T_1 -> T_2`. Parentheses can be used for type grouping. For example $(A \times B) \times C$ represents the type of pairs where the first projection is a pair of type $A \times B$ and the second projection is of type C. Some complex types are predefined: the type $\mathcal{STRING}$ corresponds to $\mathbb{N}^*$. The type of matrixes of real numbers is $\mathcal{M}(\mathbb{R})$. More complex types can be named for more convenient notation. For example we may define the type of regular relations - which are sets of pairs of strings - as

$\mathcal{REGREL}$ **is** $2^{\mathcal{STRING} \times \mathcal{STRING}}$; `REGREL is 2^(STRING * STRING);`.

The type of an object is specified with an 'in'-statement as in

$A, B \in \mathcal{REGREL}$; `A, B in REGREL;`.

Any two objects of the same type can be checked for equality/inequality in the usual way: $A = B$ and $A \neq B$ `A ~= B`. Numerical expressions can be compared as usual: $A < B$, $A \leq B$ `A <= B`, $A > B$, $A \geq B$ `A >= B`.

Identifiers, constants, and simple terms

The most basic expressions to name objects are identifiers and constants.

Identifiers in *C(M)* have to start with a letter, can contain letters and digits and can end with apostrophes. Identifiers may also have indices, which are again identifiers. Greek letter names used in the plain text version are displayed in the mathematical layout as the corresponding Greek symbols. For example, possible identifiers of a more complex kind in display and textual form are

$X'_3, \Psi_{F''}, P_{\lambda_0}$ `X'_3, Psi_F'', P_lambda_0`

In an induction statement the inductively defined identifiers are followed by the index of the induction step written in parentheses, as in the expressions

$A^{(0)}, A^{(k+1)}, F'_{\Delta}{}^{(n+1)}$ `A@0, A@k+1, F'_Delta@n+1`

See also program Lines 8, 10, 18, 20 in Program 7.1.2.

Constants. In *C(M)* there are constants for:

Booleans	*true*, *false*
Natural numbers	301, 2014
Real numbers	−25.349, 2.1234E−23
Strings	"Example", "This is a sentence."

Simple terms. *C(M)* allows the grouping of identifiers/constants into tuples for supporting parallel assignments or multiple definitions. Tuples of identifiers/constants can be recursively grouped into subterms. Examples are:

$(a^{(k)}, b^{(k)})$	`(a@k,b@k)`
$(a, ((1, c', SE1), f_1))$	`(a,((1,c',SE1),f_1))`

If T is a tuple we obtain its projections using the predefined Proj operator (cf. Definition 1.1.9). The second projection of T is $\text{Proj}_2(T)$, written `Proj(2,T)`. We can specify the pair consisting of the second and fourth element of T as $\text{Proj}_{(2,4)}(T)$, written `Proj((2,4),T)`. Simple terms can be used for running arguments in set builder and quantifier expressions. The term can contain constants for element building and for constraining running arguments. An example is the assignment

$$M := \{(a, 0, y) \mid (a, 0, y) \in S \;\&\; a > y\}$$

```
M := {(a,0,y) | (a,0,y) in S & a > y}
```

Complex terms

Set construction. There are several operators for set construction. Sets can be constructed by simply listing the elements as in $\{a, b, c\}$ (written `{a,b,c}`), or by specifying a range of numbers as in $\{n_1, \ldots, n_2\}$ `{n_1..n_2}`. Similarly as in standard mathematical notation sets can be specified with set abstraction as in

$\{f(x, z) \mid (x, y) \in A, z \in B\}$	`{f(x,z) | (x,y) in A, z in B}`

Here the simple term (x, y) runs through the elements of the finite set A. In case that e.g. y is already defined before, it will act as a constraint – only those x will be considered for which $(x, y) \in A$. A set builder can be augmented with an additional condition as in

$$\{f(x, z) \mid (x, y) \in A, z \in B \;\&\; x + y \leq z\}$$

```
{f(x,y) | (x,y) in A, z in B & x+y <= z}
```

The usual set operations union $A \cup B$, intersection $A \cap B$, set difference $A \setminus B$ are provided (written `A \/ B`, `A /\ B`, `A \ B`). If A is a set of sets, then the union of its elements is $\bigcup(A)$, written `union(A)`. The Cartesian product of the sets A, B and C is $A \times B \times C$ (written `A * B * C`).

Relations. Subsets of Cartesian products are relations. As for tuples, if R is a relation, then we can get its projections with the Proj operator (cf. Definition 1.1.9). The second projection of R is $\text{Proj}_2(R)$, written `Proj(2,R)`. We can specify the sub-relation of the second and fourth coordinates of R as $\text{Proj}_{(2,4)}(R)$. Since relations are sets of tuples they can be constructed with the set builder construction as well.

Membership and non-membership check of an element a in a set (or list) A can be expressed as $a \in A$ `a in A` and $a \notin A$ `a ~in A`. Inclusion check of a set A in B is denoted as $A \subseteq B$ and $A \not\subseteq B$ `A subseteq B`, `A ~subseteq B`.

List construction. Lists are constructed in an analogous way – by simply listing elements as in $\langle a, b, c\rangle$ (written `[a,b,c]`), by specifying a range of numbers as in $\langle 1, \ldots, n_2\rangle$ `[1..n_2]`, or using list builders as in $\langle f(x, z) \mid (x, y) \in A, z \in B\rangle$. The latter form can again be augmented with an additional condition as in

$$\langle f(x, z) \mid (x, y) \in A, z \in B \;\&\; x + y \leq z\rangle$$

```
[f(x,y) | (x,y) in A, z in B & x+y <= z]
```

Concatenation of two lists A and B is denoted $A.B$. If A is a list of lists, then $\bigodot(A)$ (written `flatten(A)`) is the concatenation of the elements of A. The number of elements in a set or a list A is denoted as $|A|$. The i$^{\text{th}}$ element of a list L is $(L)_i$, written `L[i]`. The sublist from the i$^{\text{th}}$ to the j$^{\text{th}}$ element of a list L is $(L)_{i,\ldots,j}$, written `L[i..j]`. By $\#_L(a)$ (written `#(a,L)`) we denote the index of a in L. If a is not an element of L, then 0 is returned. If there are more than one occurrences of a in L, then one of the corresponding indices is returned. The set of elements of a list L is set(L). The minimal element, maximal element, and the sum and product of the elements of a set or a list A of appropriate type are respectively denoted as $\min(A)$, $\max(A)$, $\sum(A)$ `sum(A)`, and $\prod(A)$ `prod(A)`.

Functions. Functions in *C(M)* are either finite or specified with an expression over the arguments as in

$$f(x, y) := x + 2 \times y; \qquad \texttt{f(x,y) := x+2*y;}$$

Finite functions are sets of pairs and can be constructed with the usual set and relation constructors. If f is a function, then as usual $f(a)$ denotes the image of a, and $!f(a)$ checks whether f is defined for a.

Lifting: If f is a function of type $T_1 \to T_2$ and A is a set of type 2^{T_1}, then $f(A)$ is the image of the set A under f.

Currying: If f is a function of type $T_1 \times T_2 \to T_3$ and a is of type T_1, then $f(a)$ is a function of type $T_2 \to T_3$, such that $f(a)(b) = f(a, b)$. Functions (both finite and infinite) are treated as normal objects and can be passed as parameters, returned by other functions, added as elements in sets etc.

Folding: If f is a function of type $T \times T \to T$ and $A = \langle t_1, t_2, \ldots, t_n\rangle$ is a list of type T^*, then $f(A) = f(f(\ldots f(f(t_1, t_2), t_3), \ldots, t_{n-1}), t_n)$.

Functionalizing: If R is a relation of type $2^{T_1 \times T_2}$, then we can "functionalize" it using the Func operator. Following Definition 1.1.16, $\text{Func}_{1\to 2}(R)$ (written `Func(1,2,R)`) denotes the finite function

$$\{(a, \{b \,|\, (a, b) \in R\}) \,|\, a \in Proj_1(R)\}.$$

More generally, if R is an n-ary relation and $\{i_1, \ldots, i_k\}, \{j_1, \ldots, j_l\}$ are nonempty subsets of $\{1, \ldots, n\}$, then $Func_{(i_1,\ldots,i_k)\to(j_1,\ldots,j_l)}(R)$ (written in tuple notation `Func((i_1,...,i_k),(j_1,...,j_l),R)`) is the function with domain

$Proj_{(i_1,\ldots,i_k)}(R)$ that maps an element $(a_{i_1},\ldots,a_{i_k})$ to the set

$$\{(a_{j_1},\ldots,a_{j_l})|(a_1,\ldots a_n)\in R\}.$$

Arithmetic expressions. The usual arithmetic expressions are provided for the numerical types – addition $a+b$, subtraction $a-b$, negative value $-a$, absolute value $|a|$, multiplication $a\times b$ (written `a*b`), division a/b, power a^b (written `a^b`) and reminder $a \operatorname{rem} b$.

Boolean expressions include

- negation $\neg P$, `~P`
- conjunction $P\wedge Q$, `P /\ Q`
- disjunction $P\vee Q$, `P \/ Q`
- implication $P\to Q$, `P -> Q`
- equivalence $P\leftrightarrow Q$, `P <-> Q`

In addition *C(M)* supports bounded quantifiers like

$\forall x\in X:(\exists y\in Y:(x>y))$ `forall x in X:(exists y in Y:(x>y))`

Conditional expressions can be specified as in:

$$f(a):=\begin{cases} a-1 & \textbf{if } a>5\\ a & \textbf{if } (a>0)\wedge(a\leq 5)\\ 0 & \textbf{otherwise;}\end{cases}$$

```
f(a) := ?  a-1 if a > 5
        ?  a if (a>0) /\ (a <= 5)
        ?  0 otherwise;
```

Matrix calculation. A one row matrix specified as $[a,b,c]$, written `[:a,b,c:]`. Rows and matrices can be appended downwards like `[:1,2,3:]\\[:4,5,6:]` to construct the matrix

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}.$$

Another way to construct matrices is by using a matrix builder: $[f(i,j)\mid i=1,\ldots,2,\ j=1,\ldots,3]$, written `[:f(i,j) | i=1..2, j=1..3:]`, where f is an expression of type $\mathbb{N}\times\mathbb{N}\to\mathbb{R}$.

If M is a matrix, then $M_{i,j}$ `M[i,j]` is the i$^{\text{th}}$ row j$^{\text{th}}$ column element of M. The usual matrix operations like addition, subtraction and multiplication (written `A+B`, `A-B`, `A*B` resp.) are implemented. Transposition is denoted as M^T `M^T`.

Statements

Each *C(M)* program is a list of statements. There are four kinds of statements – type definitions, declarations, assignments and supplementary actions. Every statement must end with a semicolon.

Type definitions. A type definition is used for naming a complex type. For example in Line 1 of Program 7.1.2 we have the following type definition:

$$\mathcal{REL} \textbf{ is } 2^{\mathbb{N}\times\mathbb{N}}; \quad \boxed{\texttt{REL is 2\^{}(IN*IN);}}$$

Declarations. Declaration statements are used for declaring the types of identifiers and terms used. In cases where the type of an object can be directly derived from the expression the declaration can be omitted. But in most cases when an object is defined by induction or when a function is defined the inference of the exact type is generally difficult and a type declaration is needed. The statement in Line 7 of Program 7.1.2 declares the type of the function "compose":

$$\text{compose} : \mathcal{REL} \times \mathcal{REL} \to \mathcal{REL}; \quad \boxed{\texttt{compose in REL * REL -> REL;}}$$

Assignments are the most commonly used statements in *C(M)*. There are four types of assignments – simple assignments, assignments with 'where' block, case assignments and inductive assignments.

Simple assignments. Simple assignments consist of an expression only. They are used to define the value of a term. Line 8 of Program 7.1.3 gives an example:

$$(a, b) := (C_R{}^{(n)} \textbf{ as } (\mathbb{N} \times \mathbb{N})^*)_{n+1};$$

Function values can be defined refering to arguments as in Line 3 of Program 7.1.2:

$$\text{compose}(R_1, R_2) := \{(a, c) \mid (a, b) \in R_1, (b, c) \in R_2\};$$

Assignments with 'where' block Assignment statements with a 'where'-block appear as simple assignments that are followed by a block of additional statements after the 'where' keyword. The additional statements define the objects used in the expression and may include further statements. The statements below give an illustration:

$$\begin{aligned} &\text{compose}_{2\mathrm{N}}(R, N) := \text{compose}(R', R'), \textbf{ where} \\ &\quad R' := \text{compose}_{\mathrm{N}}(R, n); \\ &\quad ; \end{aligned}$$

Note that the semicolon on the last line closes the whole statement – as discussed above each statement has to end with a semicolon.

Case assignment The case assignment is formed of a list of pairs consisting of a condition and an expression. The value of the expression of the first pair in which the condition is evaluated to true is assigned to the term. An "otherwise"

case expression is optional and used if none of the previous conditions holds. The case assignment differs from the conditional expression by the option to include a 'where' block for each of the cases. Instead of using a conditional expression in Line 9 of Program 7.1.3 we could alternatively use a case assignment:

$$
\begin{array}{l}
{C_R}^{(n+1)} := \\
\quad \textbf{case } !\,\mathrm{F_R}(b) : {C_R}^{(n)} \cup A', \textbf{where} \\
\quad\quad A' := \{(a,c) \mid c \in \mathrm{F_R}(b)\} \\
\quad \textbf{otherwise } : {C_R}^{(n)} \\
\quad ;
\end{array}
$$

Inductive assignment The inductive assignment is used for inductive constructions of terms. Lines 4-11 from Program 7.1.3 present an inductive assignment.

$$
\begin{array}{l}
C_R := \textbf{induction} \\
\quad \textbf{step } 0 : \\
\quad\quad {C_R}^{(0)} := R; \\
\quad \textbf{step } n+1 : \\
\quad\quad (a,b) := ({C_R}^{(n)} \textbf{ as } (\mathbb{N} \times \mathbb{N})^*)_{n+1}; \\
\quad\quad {C_R}^{(n+1)} := \begin{cases} {C_R}^{(n)} \cup \{(a,c) \mid c \in \mathrm{F_R}(b)\} & \textbf{if } !\,\mathrm{F_R}(b) \\ {C_R}^{(n)} & \textbf{otherwise;} \end{cases} \\
\quad \textbf{until } n = \left|{C_R}^{(n)}\right| \\
\quad ;
\end{array}
$$

The base of the induction is defined by the statements after the step 0 clause. The inductive step is defined by the statements after the step $n+1$ clause. The inductive steps are repeated until the condition of the 'until' clause holds.

Supplementary actions

Besides definitions, declarations and assignments, *C(M)* supports supplementary actions for input and output operations and similar issues. In contrast to all other statements, these actions can have side effects. For example we can dump out the relation resulting by the transitive closure of $\{(1,2),(2,3),(3,5),(5,10)\}$ with the following statement:

$$\text{dump} \leftarrow \text{transitiveClosure}(\{(1,2),(2,3),(3,5),(5,10)\});$$

```
dump <- transitiveClosure({(1,2),(2,3),(3,5),(5,10)});
```

Some of the implemented supplementary actions are

- dump $\leftarrow$ E; : dumps the expression E to the standard output;

- print $\leftarrow$ E; : the expression E (which must be a STRING) is printed on the standard output;
- store $\leftarrow$ (F, E); : the expression E will be stored in the file with name F (must be a STRING);
- saveText $\leftarrow$ (F, T); : the text T (must be STRING) will be stored in UTF-8 format in the text file with name F (must be a STRING);
- import $\leftarrow$ F; : the file named with the expression F (which must be a STRING) is imported in the program;
- assert $\leftarrow$ E; : if the expression E of type $\mathbb{B}$ does not hold, the program halts.

Chapter 8

C(M) implementation of finite-state devices

In this chapter we present *C(M)* implementations of the main automata constructions. Our aim is to provide full, clear and easy to follow descriptions of the implementations. In some cases the simplicity of the implementation is achieved at the expense of some inefficiency. Other implementations of automata and transducer constructions include XFST [Karttunen et al., 1997b], SFST [Schmid, 2006] and openFST [Allauzen et al., 2007].

8.1 *C(M)* implementations for automata algorithms

This section describes the *C(M)* implementation of the constructions presented in Chapter 2. Only algorithms for automata over the free monoid are presented.

General assumption. *In our constructions we always assume that the set of states of an automaton has the form* $\{1, 2, \ldots, n\}$. *All automata resulting from the constructions presented below will again have this property. If procedures are called with input automata that do not satisfy this assumptions, errors may result.*

Below all programs will be presented using the more readable "pretty-print" presentation of the code. Note that typed and executable versions of all programs are available.

C(M) programs for basic algorithms

Program 8.1.1 We start with the basic definitions and algorithms for finite-state automata. In our formalization, an automaton has a *set* of initial states, and transition labels are arbitrary *words* over the input alphabet.

```
1  import ← "Program_7.1.3.cm";
2  SYMBOL is ℕ;
3  ALPHABET is 2^SYMBOL;
4  WORD is SYMBOL*;
```

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

```
5   𝒮𝒯𝒜ℰ is ℕ;
6   𝒜𝒯ℛ𝒜𝒩𝒮𝐼𝒯𝐼𝒪𝒩 is 𝒮𝒯𝒜ℰ × 𝒲𝒪ℛ𝒟 × 𝒮𝒯𝒜ℰ;
7   ℱ𝒮𝒜 is 𝒜ℒ𝒫ℋ𝒜ℬℰ𝒯 × 2^𝒮𝒯𝒜ℰ × 2^𝒮𝒯𝒜ℰ × 2^𝒮𝒯𝒜ℰ × 2^𝒜𝒯ℛ𝒜𝒩𝒮𝐼𝒯𝐼𝒪𝒩;
8   newState : 2^𝒮𝒯𝒜ℰ → 𝒮𝒯𝒜ℰ;
9   newState(Q) := |Q| + 1;
10  kNewStates : ℕ × 2^𝒮𝒯𝒜ℰ → 𝒮𝒯𝒜ℰ*;
11  kNewStates(k, Q) := ⟨|Q| + 1, ..., |Q| + k⟩;
12  remap_FSA : ℱ𝒮𝒜 × 2^𝒮𝒯𝒜ℰ → ℱ𝒮𝒜;
13  remap_FSA((Σ, Q, I, F, Δ), Q') := (Σ, map(Q), map(I), map(F), Δ'), where
14      S := kNewStates(|Q|, Q');
15      map : 𝒮𝒯𝒜ℰ → 𝒮𝒯𝒜ℰ;
16      map(q) := (S)_q;
17      Δ' := {(map(q), c, map(r)) | (q, c, r) ∈ Δ};
18      ;
19  FSA_WORD : 𝒲𝒪ℛ𝒟 → ℱ𝒮𝒜;
20  FSA_WORD(a) := (set(a), {1, 2}, {1}, {2}, {(1, a, 2)});
```

In Line 1 we import Program 7.1.3, which will be used later in the section. In Lines 2-7 appropriate types are defined. Symbols are encoded as natural numbers, an $\mathcal{ALPHABET}$ as a set of symbols, a $\mathcal{WORD}$ as a list of symbols, and a $\mathcal{STATE}$ is a natural number. For automaton transitions we introduce the type $\mathcal{ATRANSITION}$, each transition is a triple of source state, word and destination state. The type $\mathcal{FSA}$ for automata over the free monoid is defined to be a quintuple consisting of an alphabet, a set of states, a set of initial states, a set of final states, and a set of transitions.

The function newState defined in Lines 8-9 takes a set of states (i.e., natural numbers) Q and returns the next new state outside Q. As we mentioned above we assume that $Q = \{1, \ldots, |Q|\}$. Hence the new state can be simply defined as the number $|Q| + 1$. The function kNewState defined in Lines 10-11 returns the list of the next k new states $\langle |Q| + 1, \ldots, |Q| + k \rangle$ outside a given set Q.

The function $\text{remap}_{\text{FSA}}$ defined in Lines 12-18 takes as arguments an automaton $(\Sigma, Q, I, F, \Delta)$ and a set of states Q'. The function remaps all states in the automaton to states outside Q'. In Line 14 we define S as the list with $|Q|$ new states outside Q'. In Lines 15-16 we define the function map, which maps the state q (in Q) to the q-th state of the list S. Here again we assume that $Q = \{1, \ldots, |Q|\}$. In Line 17 the set of transitions Δ' is defined as variant of Δ where states have been renamed outside Q. Then in Line 13 the resulting automaton is defined as the quintuple consisting of the alphabet Σ, the images of Q, I, F and the new set of transitions Δ'.

The function FSA_{WORD} constructs for a given word a the automaton recognizing $\{a\}$ as defined in Part 2 of Proposition 2.4.1.

Program 8.1.2 The following constructions present the regular operations **union**, **concatenation**, and **Kleene star** for finite-state automata.

```
21  union_FSA : ℱ𝒮𝒜 × ℱ𝒮𝒜 → ℱ𝒮𝒜;
22  union_FSA((Σ1, Q1, I1, F1, Δ1), A2) :=
    (Σ1 ∪ Σ2, Q1 ∪ Q2, I1 ∪ I2, F1 ∪ F2, Δ1 ∪ Δ2), where
23      (Σ2, Q2, I2, F2, Δ2) := remap_FSA(A2, Q1);
```

```
24      ;
25  concat_FSA : FSA × FSA → FSA;
26  concat_FSA((Σ1, Q1, I1, F1, Δ1), A2) :=
    (Σ1 ∪ Σ2, Q1 ∪ Q2, I1, F2, (Δ1 ∪ Δ2) ∪ F1 × {ε} × I2), where
27      (Σ2, Q2, I2, F2, Δ2) := remap_FSA(A2, Q1);
28      ;
29  star_FSA : FSA → FSA;
30  star_FSA(Σ, Q1, I1, F1, Δ1) := (Σ, Q1 ∪ {q0}, {q0}, F1 ∪ {q0}, Δ), where
31      q0 := newState(Q1);
32      Δ := (Δ1 ∪ {(q0, ε, q1) | q1 ∈ I1}) ∪ {(q2, ε, q0) | q2 ∈ F1};
33      ;
```

Lines 21-24 describe the union of finite-state automata. In Line 23 we first remap the states of the second automaton in order to avoid common states. In Line 22 the union of two finite-state automata is defined exactly as in Part 1 of Proposition 2.2.1. In a similar way the concatenation of two automata is defined in Lines 25-28, exactly following the description in Part 2 of Proposition 2.2.1. The Kleene star of an automaton is defined in Lines 29-33 in accordance with Part 3 of Proposition 2.2.1.

Program 8.1.3 The next "supplementary constructions" are given for convenience. They can be expressed by our basis functions but help to keep the description of complex automata constructions transparent. We introduce the **Kleene plus** of a given automaton, **optionality** (adding the empty word) for an automaton, the automaton recognizing a given **set of symbols**, and the automaton recognizing **all words** over a given alphabet.

```
34  plus_FSA : FSA → FSA;
35  plus_FSA(Σ, Q1, I1, F1, Δ1) := (Σ, Q1 ∪ {q0}, {q0}, F1, Δ), where
36      q0 := newState(Q1);
37      Δ := (Δ1 ∪ {(q0, ε, q1) | q1 ∈ I1}) ∪ {(q2, ε, q0) | q2 ∈ F1};
38      ;
39  option_FSA : FSA → FSA;
40  option_FSA(Σ, Q, I, F, Δ) := (Σ, Q ∪ {q0}, I ∪ {q0}, F ∪ {q0}, Δ), where
41      q0 := newState(Q);
42      ;
43  symbolSet2FSA : 2^SYMBOL → FSA;
44  symbolSet2FSA(S) := (S, {1, 2}, {1}, {2}, {(1, ⟨c⟩, 2) | c ∈ S});
45  all : ALPHABET → FSA;
46  all(Σ) := star_FSA(symbolSet2FSA(Σ));
```

Lines 34-38 define the positive Kleene closure in a similar way like the Kleene star. The only difference is that the new starting state q_0 is not final. Lines 39-42 realize optionality by adding a new initial and final state q_0 to a given automaton in order to add the empty word to its language. Lines 43-44 define the function symbolSet2FSA, which takes a set of symbols S and builds an automaton with two states recognizing S. If the set S is empty, the function symbolSet2FSA returns the automaton for the empty language. Lines 45-46 present the function all(Σ), which returns an automaton recognizing all words over a given alphabet.

C(M) programs for ε-removal and further constructions

Program 8.1.4 The *C(M)* program for **ε-removal** follows Proposition 2.5.4.

```
47  removeEpsilon_FSA : FSA → FSA;
48  removeEpsilon_FSA(Σ, Q, I, F, Δ) := (Σ, Q, ⋃(C_ε(I)), F, Δ'),  where
49      C := transitiveClosure({(q, r) | (q, a, r) ∈ Δ & a = ε}) ∪ {(q, q) | q ∈ Q};
50      C_ε := F_{1→2}(C);
51      Δ' := {(q_1, a, q_2) | (q_1, a, q') ∈ Δ & a ≠ ε, q_2 ∈ C_ε(q')};
52      ;
```

In Line 49, using Program 7.1.3, we define the transitive closure C of the ε-transitions in Δ. Then in Line 50 we construct the ε-closure by functionalizing the relation C (cf. Definition 1.1.16). The resulting function C_ϵ maps each state $q \in Q$ to the set of all states that can be reached from q with a series of ε-transitions and corresponds to the forward ε-closure C_ε^f as defined in Definition 2.5.3. In Line 51 and 48 the new set of transitions Δ', the extended set of initial states, and the resulting automaton are defined in accordance with the construction in Proposition 2.5.4.

Program 8.1.5 The *C(M)* program for **ε-removal preserving state languages** closely follows Proposition 2.5.6.

```
53  removeEpsilonPreserving_FSA : FSA → FSA;
54  removeEpsilonPreserving_FSA(Σ, Q, I, F, Δ) := (Σ, Q, I, ⋃(C'_ε(F)), Δ'),
            where
55      C :=
            transitiveClosure({(q, r) | (q, a, r) ∈ Δ & a = ε}) ∪ {(q, q) | q ∈ Q};
56      C_ε := F_{1→2}(C);
57      C'_ε := F_{2→1}(C);
58      Δ' := {(q'_1, α, q'_2) | (q_1, α, q_2) ∈ Δ & α ≠ ε, q'_1 ∈ C'_ε(q_1), q'_2 ∈ C_ε(q_2)};
59      ;
```

In Line 55 we define the transitive closure C of the ε-transitions in Δ. Then in Lines 56 and 57 we respectively construct the forward ε-closure C_ε and the backward ε-closure C'_ε as defined in Definition 2.5.3. In Lines 58 and 54 the new transition relation Δ', the extended set of final states, and the resulting automaton are defined in accordance with the construction in Proposition 2.5.6.

Program 8.1.6 The following program for **trimming** an automaton removes all states that are not reachable from any initial state and all states from which no final state can be reached (cf. Definition 2.5.1).

```
60  trim_FSA : FSA → FSA;
61  trim_FSA(Σ, Q, I, F, Δ) :=
    (Σ, map(Q'), map(I ∩ Q'), map(F ∩ Q'), Δ'),  where
62      R := transitiveClosure(Proj_{(1,3)}(Δ));
63      Q' := (I ∪ {r | (i, r) ∈ R & i ∈ I}) ∩ (F ∪ {r | (r, f) ∈ R & f ∈ F});
64      map : STATE → STATE;
65      map(q) := #_{Q'}  as STATE*(q);
```

```
66      Δ' := {(map(q), c, map(r)) | (q, c, r) ∈ Δ & (q ∈ Q') ∧ (r ∈ Q')};
67      ;
```

In Line 62 we define the set R consisting of all pairs of states that are connected by a path of length ≥ 1. Here again we make use of Program 7.1.3. In Line 63 we define the set Q' of all states which are reachable from an initial state and from which at the same time a final state can be reached. In Lines 64-65 the mapping of the states in Q' to $\{1, \ldots, |Q'|\}$ is defined. Each state $q \in Q'$ is mapped to the index of q in Q', the latter set is here treated as a list of states (Line 65). In Line 66 the new set of transitions Δ' is defined as the renamed variant of all transitions in Δ between states in Q'. In Line 61 we define the resulting automaton. The set of (resp. initial, final) states is the renamed variant of Q' (resp. $I \cap Q'$, $F \cap Q'$) and the set of transitions Δ'.

Program 8.1.7 Given an arbitrary finite-state automaton, our next program constructs a **classical** finite-state automaton, i.e., an automaton where each transition label is a word of length ≤ 1. Following Proposition 2.5.8 we expand each transition of the form $\langle q, a_1 a_2 \ldots a_l, r\rangle$ with a transition label of length $l > 1$ into a sequence of l transitions $\langle q' = t_1, a_1, t_2\rangle \langle t_2, a_2, t_3\rangle \ldots \langle t_l, a_l, t_{l+1} = r\rangle$ by introducing $l - 1$ new intermediate states $t_2, t_3, \ldots, t_l$.

```
68  expand_FSA : FSA → FSA;
69  expand_FSA(Σ, Q, I, F, Δ) := (Σ, Q', I, F, Δ'), where
70      L := ⟨(q', a', r') | (q', a', r') ∈ Δ & |a'| > 1⟩;
71      (Q', Δ') := induction
72          step 0 :
73              (Q'^(0), Δ'^(0)) := (Q, Δ);
74          step n + 1 :
75              (q, a, r) := (L)_{n+1};
76              l := |a|;
77              t := ⟨q⟩ · kNewStates(l − 1, Q'^(n)) · ⟨r⟩;
78              Q'^(n+1) := Q'^(n) ∪ set(t);
79              Δ'^(n+1) := Δ'^(n) \ {(q, a, r)} ∪ {((t)_i, ⟨(a)_i⟩, (t)_{i+1}) | i ∈ {1, ..., l}};
80          until n = |L|
81          ;
82      ;
```

The algorithm builds the resulting automaton by inductively defining the new set of states Q' and the new set of transitions Δ'. At each step we eliminate one transition $(q, a, r) \in \Delta$ such that $|a| > 1$. First, in Line 70 we extract from Δ the list L of such transitions. The induction starts with the original set of states and transitions in Line 73. In the inductive step in Line 75 we select the next transition (q, a, r) from the list L. In Line 77 we define t as the sequence of states with first element q, followed by $l - 1$ new states and ending with r. In Line 78 the set $Q'^{(n+1)}$ is defined as the union of $Q'^{(n)}$ with the states in t. In the transition set Δ' we replace (q, a, r) by the transitions (t_i, a_i, t_{i+1}) for $i = 1, \ldots, l$ (Line 79). The induction ends when the list L is exhausted. The resulting automaton is then defined as the automaton with set of states Q' and set of transitions Δ'. Initial and final states are as in the source automaton

(Line 69).

C(M) programs for determinization, intersection and difference of automata

In our description, the type of *deterministic* finite-state automata is distinct from the type of general finite-state automata: deterministic finite-state automata have a single initial state (as opposed to a set of initial states) and a transition function (as opposed to a relation). We now introduce the new types needed and a variant of the trimming construction for deterministic finite-state automata.

Program 8.1.8 As in the general case (cf. Program 8.1.6), **trimming** a deterministic automaton means to remove all states that are not reachable from the initial states and all states from which no final state can be reached (Definition 2.5.1).

```
83  DTRANSITIONS is STATE × SYMBOL → STATE;
84  DFSA is ALPHABET × 2^STATE × STATE × 2^STATE ×
    DTRANSITIONS;
85  trim_DFSA : DFSA → DFSA;
86  trim_DFSA(Σ, Q, q_0, F, Δ) :=
    { (Σ, map(Q'), map(q_0), map(F ∩ Q'), Δ')   if Q' ≠ ∅
    { (Σ, {1}, 1, ∅, ∅)                         otherwise   , where
87      R := transitiveClosure({(s, d) | ((s, a), d) ∈ Δ});
88      Q' := ({q_0} ∪ {r | (q_0, r) ∈ R}) ∩ (F ∪ {r | (r, f) ∈ R & f ∈ F});
89      map : STATE → STATE;
90      map(q) := #_Q' as STATE*(q);
91      Δ' := {((map(q), c), map(r)) | ((q, c), r) ∈ Δ & (q ∈ Q') ∧ (r ∈ Q')};
92      ;
```

In Line 83 the type $\mathcal{DTRANSITIONS}$ for the transition function of a deterministic finite-state automaton is defined. Then in Line 84 a $\mathcal{DFSA}$ is defined as a quintuple consisting of an alphabet, a set of states, an initial state, a set of final states, and a transition function. The trim function $\text{trim}_{\text{DFSA}}$ defined in Lines 85-92 is almost identical to the corresponding function for nondeterministic automata (Program 8.1.6). The only difference is in Line 86, where we now return a deterministic automaton with one start state instead of an empty automaton.

Program 8.1.9 In Theorem 3.2.2 we presented a simple determinization construction for finite-state automata. This construction always builds an automaton with $2^{|Q|}$ states, not taking into account that many of the states may not be reachable from the resulting new initial state. Below we present a more efficient **determinization** construction. Following Remark 3.2.3 it builds a deterministic automaton where all states are reachable from the new initial state. As a first preparation step it computes a non-deterministic automaton where all transition labels have length 1. Then the deterministic automaton is built inductively, starting from the set of initial states (acting as the new initial state).

While the construction proceeds, a list of new states introduced is maintained. Each new state is a set of old states. At each induction step we treat the next element in the list and compute, for each symbol σ of the input alphabet, the sets of states which are reachable from the current set with letter σ. Each new set of states obtained in this way is added as a new state to the tail of the list of new states. The induction ends when all sets in the list have been treated.

```
 93  determ_FSA : FSA → DFSA;
 94  determ_FSA(A) := trim_DFSA(Σ, {1,...,|P|}, 1, F', δ'),  where
 95      (Σ, Q, I, F, Δ) := expand_FSA(removeEpsilon_FSA(A));
 96      Δ' := F_{1→(2,3)}({(q, (α)_1, r) | (q, α, r) ∈ Δ});
 97      (P, δ') ∈ (2^{STATES})* × TRANSITIONS;
 98      (P, δ') := induction
 99          step 0 :
100              P^(0) := ⟨I⟩;
101              δ'^(0) := ∅;
102          step n + 1 :
103              N := ⋃(Δ'((P^(n))_{n+1}));
104              N' := F_{1→2}(N);
105              P^(n+1) := P^(n) · ⟨q | q ∈ Proj_2(N') & q ∉ P^(n)⟩;
106              δ'^(n+1) := δ'^(n) ∪ {((n + 1, c), #_{P^(n+1)}(q)) | (c, q) ∈ N'};
107          until n = |P^(n)|
108          ;
109      F' := {q | q ∈ {1,...,|P|} & F ∩ (P)_q ≠ ∅};
110      ;
```

The preliminary first step is described in Line 95: ε-transitions are removed using Program 8.1.4, and transitions labels of length ≥ 2 are replaced by sequences of transitions with labels of length 1 using Program 8.1.7. Then the main step starts. In Line 96 we define Δ' as the functionalization $1 \to (2,3)$ of Δ (cf. Definition 1.1.16). Δ' maps a state $q \in Q$ to the set of pairs (a, r), such that $(q, a, r) \in \Delta$. Only the source states of transitions appear in the domain of Δ', hence images are always non-empty sets. The use of Δ' helps to avoid yet another induction that runs over the symbols of the alphabet. In Lines 97-108 the states of the deterministic automaton are constructed by induction. Each new state is a set of states of the source automaton (in the final version, each new state is translated into a natural number, see below). New states to be introduced are maintained in the list P. In parallel the transition function δ' is constructed. In Lines 100-101 the base of the induction is defined. $P^{(0)}$ contains only the set I (new initial state) and $\delta'^{(0)}$ is the empty function. In the inductive step in Line 103 we first take the next ($n+1$-st) element $(P^{(n)})_{n+1}$ from the current list $P^{(n)}$. N is defined as the set of all pairs (c, q) found in the images of elements of $(P^{(n)})_{n+1}$ under Δ'. At this point, if $(P^{(n)})_{n+1}$ is empty or Δ' is undefined for all elements of $(P^{(n)})_{n+1}$ we obtain the empty set. In Line 104 N' is defined as the function which maps a label c to the set of states which are reached with a c-transition from states in $(P^{(n)})_{n+1}$. In Line 105 we add to the current list of new states $P^{(n)}$ all state sets in the range of N' that yet do not occur in $P^{(n)}$. In Line 106 the function δ' is extended with the transitions from the current set of states to the corresponding sets defined with N'. At

this point each set of states (new state) is mapped to its index in the list P. In this way, new states are now numbers. The induction ends when the list P is exhausted (Line 107). In Line 109 the set of final states F' is defined as the set of the indices of sets with non empty intersection with F. Finally in Line 94 the resulting automaton is defined. The set of states in the translated representation is the set of indices $\{1, \ldots, |P|\}$, the initial state is 1. The automaton is trimmed using Program 8.1.8. In this way, states are eliminated from which we cannot reach any final state.

The constructions for intersection and difference of automata proceed by building the Cartesian product automaton (cf. Proposition 3.3.2). We again optimize the process by inductively constructing only those (pairs of) states which can be reached from the new initial state. For both constructions we start from the pair of the two initial states of the input automata (new initial state) and then proceed by extending the automaton with the states and transitions which are directly reachable from states already obtained. The following auxiliary construction is used.

Program 8.1.10 The following algorithm (product_Δ) constructs the transition function of an automaton which is the Cartesian product of two input automata. We assume that the source automata are deterministic. The arguments are the initial states and the transition functions of the two input automata. The result is the list of pairs of states and the transition function of the resulting automata. Here again the states of the resulting automaton are finally represented using the indices of the pairs of states in the list.

111 $\text{product}_\Delta : (\mathcal{STATE} \times \mathcal{DTRANSITIONS}) \times (\mathcal{STATE} \times \mathcal{DTRANSITIONS}) \to (\mathcal{STATE} \times \mathcal{STATE})^* \times \mathcal{DTRANSITIONS}$;
112 $\text{product}_\Delta((s_1, \delta_1), (s_2, \delta_2)) := (P, \delta)$, **where**
113 $\quad \Delta_1' := \mathcal{F}_{1\to(2,3)}(\delta_1$ **as** $2^{\mathcal{STATE}\times\mathcal{SYMBOL}\times\mathcal{STATE}})$;
114 $\quad (P, \delta) \in (\mathcal{STATE} \times \mathcal{STATE})^* \times \mathcal{DTRANSITIONS}$;
115 $\quad (P, \delta) :=$ **induction**
116 $\quad\quad$ **step** 0 :
117 $\quad\quad\quad P^{(0)} := \langle (s_1, s_2) \rangle$;
118 $\quad\quad\quad \delta^{(0)} := \emptyset$;
119 $\quad\quad$ **step** $n+1$:
120 $\quad\quad\quad (p_1, p_2) := (P^{(n)})_{n+1}$;
121 $\quad\quad\quad N :=$
$$\begin{cases} \{(c, (q_1, \delta_2(p_2, c))) \mid (c, q_1) \in \Delta_1'(p_1) \;\&\; !\,\delta_2(p_2, c)\} & \textbf{if } !\,\Delta_1'(p_1) \\ \emptyset & \textbf{otherwise;} \end{cases}$$
122 $\quad\quad\quad P^{(n+1)} := P^{(n)} \cdot \langle p \mid p \in \text{Proj}_2(N) \;\&\; p \notin P^{(n)} \rangle$;
123 $\quad\quad\quad \delta^{(n+1)} := \delta^{(n)} \cup \{((n+1, c), \#_{P^{(n+1)}}(q)) \mid (c, q) \in N\}$;
124 $\quad\quad$ **until** $n = \left|P^{(n)}\right|$
125 $\quad\quad$;
126 $\quad$;

In Line 113 of the construction we define Δ_1' as the function that maps a state from the first source automaton to a set of pairs of label symbol and destination state in the transition function δ_1. (As above the use of Δ' helps to avoid

another induction that runs over the symbols of the alphabet.) Then, the list of pairs of states P and the resulting transition function δ are constructed by induction. In the base step in Lines 117-128, P is set to the list with the only pair (s_1, s_2), and δ is empty. Then, in the inductive step we consider (p_1, p_2) – the $n+1$-st element of the list P. The set N defined in Line 121 describes the set of all transitions departing from the source (product-) state (p_1, p_2) as a set of pairs. Each pair consists of a transition label and a target (product-) state. If $\Delta_1'(p_1)$ is defined, then N is the set of pairs $(c, (q_1, q_2))$ such that (c, q_1) in $\Delta_1'(p_1)$ (which means that $q_1 = \delta_1(p_1, c)$) and $\delta_2(p_2, c) = q_2$ is defined. Otherwise N is empty. In Line 122 we add to the list P the pairs of states in the second projection of N that are not found in the current set $P^{(n)}$. In Line 123 the function δ is extended with the transitions from the current pair of states to the corresponding target pairs defined in N. Here the pairs of states are mapped to their corresponding indices in the list P. The induction ends when the list P is exhausted (Line 124).

The above function represents the essential part of the programs for intersection and difference to be described now.

Program 8.1.11 The following *C(M)* program implements **intersection** of deterministic classical finite-state automata as described in Part 1 of Proposition 3.3.2. Another program for intersecting arbitrary classical finite-state automata is derived and added.

```
127  intersect_DFSA : DFSA × DFSA → DFSA;
128  intersect_DFSA((Σ1, Q1, s1, F1, δ1), (Σ2, Q2, s2, F2, δ2)) :=
     trim_DFSA(Σ1 ∪ Σ2, Q, 1, F, δ), where
129      (P, δ) := product_Δ((s1, δ1), (s2, δ2));
130      Q := {1, ..., |P|};
131      F := {q | q ∈ Q & (Proj1((P)q) ∈ F1) ∧ (Proj2((P)q) ∈ F2)};
132      ;
133  FSA_DFSA : DFSA → FSA;
134  FSA_DFSA(Σ, Q, q0, F, δ) := (Σ, Q, {q0}, F, {(q, ⟨a⟩, r) | ((q, a), r) ∈ δ});
135  intersect_FSA : FSA × FSA → FSA;
136  intersect_FSA(A1, A2) :=
     FSA_DFSA(intersect_DFSA(determ_FSA(A1), determ_FSA(A2)));
```

The function $\text{intersect}_{\text{DFSA}}$ constructs the intersection of two input automata. In Line 129 the Cartesian product of the two automata, (P, δ), is constructed using Program 8.1.10. Then in Line 130 the states of the automaton are defined as the set $\{1, \ldots, |P|\}$. In Line 131 the final states are defined as the indices of the state pairs for which both states are final in their corresponding automata. After trimming in Line 128 the automaton is returned.

The remaining code represents the intersection program for non-deterministic automata. In Lines 133-134 we define the function FSA_{DFSA}, which converts a deterministic finite-state automaton to an arbitrary finite-state automaton. In Lines 135-136 the function $\text{intersect}_{\text{FSA}}$ is defined. This function intersects two non-deterministic automata by first determinizing the input automata (Program 8.1.9) and then constructing the intersection using the above function

for deterministic automata. The resulting deterministic automaton is finally converted to a non-deterministic automaton.

Program 8.1.12 The next *C(M)* program implements **difference** of finite-state automata as described in Part 2 of Proposition 3.3.2.

```
137  diff_DFSA : DFSA × DFSA → DFSA;
138  diff_DFSA((Σ1, Q1, s1, F1, δ1), (Σ2, Q2, s2, F2, δ2)) :=
     trim_DFSA(Σ1 ∪ Σ2, Q, 1, F, δ),  where
139      δ2' ∈ TRANSITIONS;
140      δ2'(q, c) := { δ2(q, c)   if ! δ2(q, c)
                      { 0         otherwise;
141      (P, δ) := product_Δ((s1, δ1), (s2, δ2'));
142      Q := {1, ..., |P|};
143      F := {q | q ∈ Q & (Proj_1((P)_q) ∈ F1) ∧ (Proj_2((P)_q) ∉ F2)};
144      ;
145  diff_FSA : FSA × FSA → FSA;
146  diff_FSA(A1, A2) := FSA_DFSA(diff_DFSA(determ_FSA(A1), determ_FSA(A2)));
```

The first function $\text{diff}_{\text{DFSA}}$ constructs the difference of two deterministic automata. Since the second automaton has to be total (see Proposition 3.3.2), we extend in Line 140 the transition function δ_2 by complementing it with transitions to a fail state (0). Then in Line 141 the Cartesian product of the two automata (P, δ) is constructed using again Program 8.1.10. In Line 142 the set of states of the automaton is defined as $\{1, \ldots, |P|\}$. In Line 143 the final states are defined to be the indices of those pairs of states for which the state of the first automaton is final and the state in the second automaton is not final. After trimming in Line 138 the automaton is returned as result.

In Lines 145-146 the function diff_{FSA} is defined. This function computes the difference of two non-deterministic automata by first determinizing the automata (Program 8.1.9) and then constructing the difference using the above function for deterministic automata. The resulting deterministic automaton is converted to a non-deterministic one at the end.

Program 8.1.13 The following *C(M)* program implements **reversal** of finite-state automata as described in Proposition 3.3.4.

```
147  ρ : WORD → WORD;
148  ρ(s) := ⟨(s)_{|s|−i+1} | i ∈ {1, ..., |s|}⟩;
149  reverse_FSA : FSA → FSA;
150  reverse_FSA(Σ, Q, I, F, Δ) := (Σ, Q, F, I, Δ'),  where
151      Δ' := {(q2, ρ(w), q1) | (q1, w, q2) ∈ Δ};
152      ;
```

First in Lines 147-148 the function ρ for reversing a word is given. Then the construction of the reverse automaton in Lines 149-152 closely follows the description given in Proposition 3.3.4

C(M) programs for minimizing deterministic automata

In this subsection we present an algorithm which - given a deterministic finite-state automaton - constructs the equivalent minimal deterministic automaton as characterized in Section 3.4. The presented algorithm is designed for deterministic automata with partial transition function as discussed in Remarks 3.4.19 and 3.5.8

Program 8.1.14 The algorithm implements a **minimization** procedure for deterministic finite-state automata based on the inductive construction presented in Corollary 3.5.4, using the functions defined in Proposition 3.5.6.

```
153  ℰ𝒬ℛℰℒ is 𝒮𝒯𝒜𝒯ℰ → 𝒮𝒯𝒜𝒯ℰ;
154  ker : 2^(𝒮𝒯𝒜𝒯ℰ) × (𝒮𝒯𝒜𝒯ℰ → ℕ) → ℰ𝒬ℛℰℒ;
155  ker(Q, g) := {(q, #_I as ℕ*(g(q))) | q ∈ Q}, where
156      I := {g(q) | q ∈ Q};
157      ;
158  intersect_EQREL : 2^(𝒮𝒯𝒜𝒯ℰ) × ℰ𝒬ℛℰℒ × ℰ𝒬ℛℰℒ → ℰ𝒬ℛℰℒ;
159  intersect_EQREL(Q, R1, R2) := {(q, #_I as (𝒮𝒯𝒜𝒯ℰ×𝒮𝒯𝒜𝒯ℰ)*((R1(q), R2(q)))) | q ∈ Q}, where
160      I := {(R1(q), R2(q)) | q ∈ Q};
161      ;
162  minimal_DFSA : 𝒟ℱ𝒮𝒜 → 𝒟ℱ𝒮𝒜;
163  minimal_DFSA(Σ, Q, q0, F, δ) := trim_DFSA(Σ, R(Q), R(q0), R(F), δ'), where
164      R ∈ ℰ𝒬ℛℰℒ;
165      k ∈ ℕ;
166      (R, k) := induction
167          step 0 :
168              f : 𝒮𝒯𝒜𝒯ℰ → ℕ;
169              f(q) := { 1  if q ∈ F
                         { 0  otherwise;
170              k^(0) := 0;
171              R^(0) := ker(Q, f);
172          step n + 1 :
173              f : 𝒮𝒴ℳℬ𝒪ℒ × 𝒮𝒯𝒜𝒯ℰ → ℕ;
174              f(c, q) := { R^(n)(δ(q, c))  if !δ(q, c)
                            { 0              otherwise;
175              k^(n+1) := |Proj_2(R^(n))|;
176              R^(n+1) := intersect_EQREL(Q, R^(n),
                     (intersect_EQREL(Q))(⟨ker(Q, f(c)) | c ∈ Σ⟩));
177          until k^(n) = |Proj_2(R^(n))|
178          ;
179      δ' ∈ 𝒟𝒯ℛ𝒜𝒩𝒮𝐼𝒯𝐼𝒪𝒩𝒮;
180      δ' := {((R(q), σ), R(r)) | ((q, σ), r) ∈ δ};
181      ;
```

In Line 153 we define the type of an equivalence relation as a function mapping a state to (the number of) its equivalence class. We assume that the equivalence classes are numbered sequentially starting from 1. In Lines 154-157 we construct the kernel equivalence relation for a function on the automaton states. In Lines 158-159 we construct the intersection of two equivalence relations. The

equivalence class of the intersection for a given state q is defined as the index of the pair of the two equivalence classes of q with respect to the two source relations in the list I of all pairs of classes. Afterwards in Lines 163-181 we present the minimization algorithm. We construct the equivalence relation R by induction. In $k^{(n+1)}$ we store the number of equivalence classes of the relation in the previous step – $R^{(n)}$. The base of the induction is defined in Lines 168-171 in accordance with Proposition 3.5.6, Point 1. For the inductive step in Lines 173-176 we use the definition given in Proposition 3.5.6, Point 2. Note that in Line 176 the function $(\text{intersect}_{\text{EQREL}}(Q))$ is obtained by Currying, it has type $\mathcal{EQREL} \times \mathcal{EQREL} \to \mathcal{EQREL}$. This function is applied to the list of letter-specific equivalence relations $\langle \ker(Q, \mathrm{f}(c)) \mid c \in \Sigma \rangle$. The result is obtained by folding (see Section 7.2). The resulting equivalence relation is intersected with the $R^{(n)}$, which yields $R^{(n+1)}$. The induction ends when the number of classes of $R^{(n)}$ is equal to $k^{(n)}$, which is the number of classes of $R^{(n-1)}$ (Line 177). In Line 179-180 we map the states of the transitions to the numbers of their equivalence classes with respect to R and in Line 163 the automaton states, the start state and the final states are mapped to the numbers of their equivalence classes. The algorithm returns the resulting automaton after trimming.

C(M) programs for traversing deterministic finite-state automata

Program 8.1.15 Below we implement two basic functions on deterministic finite-state automata. The first function 'C_δ' implements the transitive closure of the transition function δ^*. The second function 'stateseq' returns the sequence of states on the automaton path starting from the given input state that is labeled with the given word.

```
182  C_δ : DTRANSITIONS × STATE × WORD → STATE;
183  C_δ(δ, q, α) := q′, where
184      q′ := induction
185          step 0 :
186              q′^(0) := q;
187          step n + 1 :
188              q′^(n+1) := δ(q′^(n), (α)_{n+1});
189          until n = |α|
190          ;
191      ;
192  stateseq : DTRANSITIONS × STATE × WORD → STATE*;
193  stateseq(δ, q, α) := π, where
194      π := induction
195          step 0 :
196              π^(0) := ⟨q⟩;
197          step n + 1 :
198              π^(n+1) := π^(n) · ⟨δ((π^(n))_{n+1}, (α)_{n+1})⟩;
199          until (n = |α|) ∨ ¬(! δ((π^(n))_{n+1}, (α)_{n+1}))
200          ;
201      ;
```

Both algorithm proceed by a simple inductive construction. Starting from the given first state q as a base (Line 186/196) we proceed to the next state by making the transition with the following symbol of the input word α (Line 188/198). The induction ends if all the symbols from α are consumed (Line 189/199) or in stateseq if the transition with the following symbol is not defined (Line 199). Note that 'C_δ' is not defined if for some state on the path the transition function with the following symbol from α is not defined. In contrast, 'stateseq' is always defined and returns the states on the longest path from q with a prefix from α.

The above program can be used for testing whether a word is accepted by the language of an deterministic automaton. If $A = (\Sigma, Q, q_0, F, \delta)$ is a total deterministic automaton and α is a word in Σ^*, then α is recognized by A iff $C_\delta(\delta, q_0, \alpha) \in F$.

Example 8.1.16 In order to illustrate the use of the algorithms presented in this section we provide a program for constructing a deterministic finite-state automaton that recognizes all valid dates of the Gregorian calendar formated as expressions of the form

```
AUGUST 11, 1996
```

We loosely follow the approach in [Karttunen et al., 1997a]. Date expressions like "`FEBRUARY 30, 2015`" or "`APRIL 31, 1921`" are easily described as incorrect. More challenging is the case with leap days. "`FEBRUARY 29, 2000`" and "`FEBRUARY 29, 2016`" are valid dates but "`FEBRUARY 29, 2017`" and "`FEBRUARY 29, 1900`" do not exist. Recall that in the Gregorian calendar, not every year divisible by four is a leap year. Exceptions (non leap years) are all numbers that are divisible by 100 but not by 400, such as e.g. 1700, 1800, 1900, 2100.

```
1  import ← "Section_8.1.cm";
2  Alphabet := {'A', ..., 'Z'} ∪ {'0', ..., '9'} ∪ {' ', ',', '};
3  OneToNine := symbolSet2FSA({'1', ..., '9'});
4  Even := symbolSet2FSA({'0', '2', '4', '6', '8'});
5  Odd := symbolSet2FSA({'1', '3', '5', '7', '9'});
6  ZeroToNine := union_FSA(Even, Odd);
7  Month29 := FSA_WORD("FEBRUARY");
8  Month30 := union_FSA(⟨FSA_WORD("APRIL"), FSA_WORD("JUNE"),
       FSA_WORD("SEPTEMBER"), FSA_WORD("NOVEMBER")⟩);
9  Month31 := union_FSA(⟨FSA_WORD("JANUARY"), FSA_WORD("MARCH"),
       FSA_WORD("MAY"), FSA_WORD("JULY"), FSA_WORD("AUGUST"),
       FSA_WORD("OCTOBER"), FSA_WORD("DECEMBER")⟩);
10 Month := union_FSA(⟨Month29, Month30, Month31⟩);
11 Date := union_FSA(⟨OneToNine,
       concat_FSA(symbolSet2FSA({'1', '2'}), ZeroToNine),
       concat_FSA(symbolSet2FSA({'3'}), symbolSet2FSA({'0', '1'}))⟩);
12 Year := concat_FSA(OneToNine, star_FSA(ZeroToNine));
13 DateExpression :=
   concat_FSA(⟨Month, FSA_WORD(" "), Date, FSA_WORD(", "),
       Year⟩);
14 MaxDays30 := diff_FSA(all(Alphabet), concat_FSA(⟨all(Alphabet),
       Month29, FSA_WORD(" 30"), all(Alphabet)⟩));
```

15 $MaxDays31 := \text{diff}_{\text{FSA}}(\text{all}(Alphabet), \text{concat}_{\text{FSA}}(\langle \text{all}(Alphabet),$
$\text{union}_{\text{FSA}}(Month29, Month30), \text{FSA}_{\text{WORD}}(\text{" 31"}), \text{all}(Alphabet)\rangle));$
16 $MaxDaysInMonth := \text{intersect}_{\text{FSA}}(MaxDays30, MaxDays31);$
17 $Div4 := \text{union}_{\text{FSA}}(\text{symbolSet2FSA}(\{'4', '8'\}),$
$\text{concat}_{\text{FSA}}(\text{star}_{\text{FSA}}(ZeroToNine),$
$\text{union}_{\text{FSA}}(\text{concat}_{\text{FSA}}(Even, \text{symbolSet2FSA}(\{'0', '4', '8'\})),$
$\text{concat}_{\text{FSA}}(Odd, \text{symbolSet2FSA}(\{'2', '6'\}))));$
18 $LeapYear := \text{diff}_{\text{FSA}}(Div4, \text{concat}_{\text{FSA}}(\text{diff}_{\text{FSA}}(\text{plus}_{\text{FSA}}(ZeroToNine),$
$Div4), \text{FSA}_{\text{WORD}}(\text{"00"})));$
19 $LeapDates := \text{diff}_{\text{FSA}}(\text{all}(Alphabet), \text{concat}_{\text{FSA}}($
$\text{FSA}_{\text{WORD}}(\text{"FEBRUARY 29, "}), \text{diff}_{\text{FSA}}(Year, LeapYear)));$
20 $ValidDates :=$
$\text{intersect}_{\text{FSA}}(\langle DateExpression, MaxDaysInMonth, LeapDates\rangle);$
21 $ValidDates_{DFSA} := \text{minimal}_{\text{DFSA}}(\text{determ}_{\text{FSA}}(ValidDates));$
22 $NonValidDates := \text{diff}_{\text{FSA}}(DateExpression, ValidDates);$
23 $NonValidDates_{DFSA} := \text{minimal}_{\text{DFSA}}(\text{determ}_{\text{FSA}}(NonValidDates));$

We start with importing all programs from Section 8.1 in Line 1. Line 2 defines the alphabet. In Lines 3-7 the finite-state automata for the digits from one to nine, for the even and odd single digit numbers and for all single digit numbers are constructed. Then in Lines 7-9 automata for three sets of month names are constructed, distinguishing months with 29, 30, and 31 days. Line 10 builds the automaton for all month names. In Lines 11 and 12 we respectively define the automata for representing all numbers from 1 to 31 and all positive numbers without leading zeros. Afterwards, in Line 13 we define the automaton *DateExpression* for all date expressions in the correct grammatical format including the non-valid ones. It remains to find a way to exclude invalid dates. This will be reached by intersecting *DateExpression* with two other automata (Line 20).

Lines 14-16 define the automaton *MaxDaysInMonth* which recognizes all strings except the ones which contain a month name followed by an inappropriate number of days.

The following program steps are devoted to the problem of recognizing leap years and taking into account that in leap years month February has 29 days. All numbers that are divisible by four are represented by the automaton constructed in Line 17. Line 18 defines the automaton for all leap years, i.e., the set of numbers divisible by 4 subtracting centuries that are not multiples of 400. Afterwards (Line 19) the automaton *LeapDates* recognizes all strings except the ones that start with "`FEBRUARY 29`" and end by a non-leap year.

Having these automata at our disposal, the automaton for the valid dates is constructed by intersecting *DateExpression* with *MaxDaysInMonth* and *LeapDates* (Line 20). The non-valid dates are defined in Line 22 by removing the valid dates from all date expressions. The corresponding minimal deterministic finite-state automata are constructed in Lines 21 and 23. The resulting automaton $ValidDates_{DFSA}$ has 72 states and 218 transitions whereas the automaton $NonValidDates_{DFSA}$ has 46 states and 140 transitions.

In Information Extraction there are many important groups of phrases that can be successfully recognized with automaton-techniques similar to those presented above. Our example of valid dates is special in the sense that here only

a small lexicon (names of months) is needed. Frequent tasks where larger lexica are used are recognition of address data (lexica of street names, cities,...), full person names (lexica of first names, family names, academic titles,...), diseases etc. Automata compiled for these applications may have millions of states. Still, also automata of this size can be built with the above operations and used without difficulties.

8.2 *C(M)* programs for classical finite-state transducers

In this section we present *C(M)* implementations for the main algorithms for n-tape automata as described in Secton 4.1. We only look at 2-tape automata over the free monoid, i.e., classical finite-state transducers. We assume that all auxiliary functions defined in Section 8.1 are imported and available.

Program 8.2.1 As a first step the types used for finite-state transducers over the free monoid are introduced, and the function for **renaming** the states of a classical finite-state transducer is defined. We then introduce a transducer for translating a given word into a second word.

```
1   import ← "Section_8.1.cm";
2   TRANSITION is STATE×(WORD×WORD)×STATE;
3   FST is ALPHABET × 2^STATE × 2^STATE × 2^STATE ×
    2^TRANSITION;
4   remap_FST : FST × 2^STATE → FST;
5   remap_FST((Σ, Q, I, F, Δ), Q') :=
    (Σ, map(Q), map(I), map(F), Δ'), where
6       S := kNewStates(|Q|, Q');
7       map : STATE → STATE;
8       map(q) := (S)_q;
9       Δ' := {(map(q), c, map(r)) | (q, c, r) ∈ Δ};
10      ;
11  FST_2WORD : WORD × WORD → FST;
12  FST_2WORD(a) :=
    (set(Proj_1(a)) ∪ set(Proj_2(a)), {1, 2}, {1}, {2}, {(1, a, 2)});
```

In Line 1 all programs from Section 8.1 are imported. For 2-tape automata, transition labels are pairs of words (Lines 2, 3). The function $\text{remap}_{\text{FST}}$ defined in Lines 4-10 is analogous to the function $\text{remap}_{\text{FSA}}$ given in Program 8.1.1. In Lines 11-12,the symbol a denotes a pair of words. The function $\text{FST}_{\text{2WORD}}$ defined in Lines 11-12 takes a pair of words a and constructs a transducer with two states and a single transition with the given pair of words as label.

Program 8.2.2 The next program describes **union**, **concatenation**, **Kleene star**, **Kleene plus**, **optionality**, and $\langle\varepsilon, \varepsilon\rangle$**-removal** for finite-state transducers. When ignoring type differences the constructions for these operations are essentially identical to the ones for 1-tape automata, cf. Programs 8.1.2, 8.1.3, 8.1.4 and 8.1.6.

```
13  union_FST : FST × FST → FST;
14  union_FST((Σ1, Q1, I1, F1, Δ1), A2) :=
    (Σ1 ∪ Σ2, Q1 ∪ Q2, I1 ∪ I2, F1 ∪ F2, Δ1 ∪ Δ2), where
15      (Σ2, Q2, I2, F2, Δ2) := remap_FST(A2, Q1);
16      ;
17  concat_FST : FST × FST → FST;
18  concat_FST((Σ1, Q1, I1, F1, Δ1), A2) :=
    (Σ1 ∪ Σ2, Q1 ∪ Q2, I1, F2, (Δ1 ∪ Δ2) ∪ F1 × {(ε, ε)} × I2), where
19      (Σ2, Q2, I2, F2, Δ2) := remap_FST(A2, Q1);
20      ;
21  star_FST : FST → FST;
22  star_FST(Σ, Q1, I1, F1, Δ1) := (Σ, Q1 ∪ {q0}, {q0}, F1 ∪ {q0}, Δ), where
23      q0 := newState(Q1);
24      Δ := (Δ1 ∪ {(q0, (ε, ε), q1) | q1 ∈ I1}) ∪ {(q2, (ε, ε), q0) | q2 ∈ F1};
25      ;
26  plus_FST : FST → FST;
27  plus_FST(Σ, Q1, I1, F1, Δ1) := (Σ, Q1 ∪ {q0}, {q0}, F1, Δ), where
28      q0 := newState(Q1);
29      Δ := (Δ1 ∪ {(q0, (ε, ε), q1) | q1 ∈ I1}) ∪ {(q2, (ε, ε), q0) | q2 ∈ F1};
30      ;
31  option_FST : FST → FST;
32  option_FST(Σ, Q, I, F, Δ) := (Σ, Q ∪ {q0}, I ∪ {q0}, F ∪ {q0}, Δ), where
33      q0 := newState(Q);
34      ;
35  removeEpsilon_FST : FST → FST;
36  removeEpsilon_FST(Σ, Q, I, F, Δ) := (Σ, Q, ⋃(C_ε(I)), F, Δ'), where
37      C := transitiveClosure({(q, r) | (q, (a1, a2), r) ∈ Δ & (a1 =
    ε) ∧ (a2 = ε)});
38      C_ε := F_{1→2}(C ∪ {(q, q) | q ∈ Q});
39      Δ' := {(q1, (a1, a2), q2) | (q1, (a1, a2), q') ∈ Δ & (a1 ≠ ε) ∨ (a2 ≠
    ε), q2 ∈ C_ε(q')};
40      ;
41  trim_FST : FST → FST;
42  trim_FST(Σ, Q, I, F, Δ) :=
    (Σ, map(Q'), map(I ∩ Q'), map(F ∩ Q'), Δ'), where
43      R := transitiveClosure(Proj_(1,3)(Δ));
44      Q' := (I ∪ {r | (i, r) ∈ R & i ∈ I}) ∩ (F ∪ {r | (r, f) ∈ R & f ∈ F});
45      map : STATE → STATE;
46      map(q) := #_Q' as ST A T E*(q);
47      Δ' := {(map(q), c, map(r)) | (q, c, r) ∈ Δ & (q ∈ Q') ∧ (r ∈ Q')};
48      ;
```

Program 8.2.3 The program below takes two 1-tape automata as input and computes a transducer representing the **Cartesian product** of the two input automaton languages as in Proposition 4.2.1, Part 1. The simple construction presented in Proposition 4.2.1 is refined, the transducer is built in an inductive way such that all states can be reached from the initial state. The construction resembles Program 8.1.10, where we computed a product-based transition function for two given deterministic finite-state automata. The latter transi-

tion function was used to compute intersection and difference of the automaton languages. Here we build a new, two-component language.

```
49  product_FSA : FSA × FSA → FST;
50  product_FSA((Σ1, Q1, I1, F1, Δ1), (Σ2, Q2, I2, F2, Δ2)) :=
    trim_FST(removeEpsilon_FST(Σ1 ∪ Σ2, {1, ..., |P|}, I, F, Δ)), where
51      E_Δ1 := Δ1 ∪ {(q, ε, q) | q ∈ Q1};
52      E_Δ2 := Δ2 ∪ {(q, ε, q) | q ∈ Q2};
53      Δ'1 := F_{1→(2,3)}(E_Δ1);
54      Δ'2 := F_{1→(2,3)}(E_Δ2);
55      (P, Δ) ∈ (STATE × STATE)* × 2^TRANSITION;
56      (P, Δ) := induction
57          step 0 :
58              P^(0) := I1 × I2 as (STATE × STATE)*;
59              Δ^(0) := ∅;
60          step n + 1 :
61              (p1, p2) := (P^(n))_{n+1};
62              D1 := Δ'1(p1);
63              D2 := Δ'2(p2);
64              N := {((a, b), (q1, q2)) | (a, q1) ∈ D1, (b, q2) ∈ D2};
65              P^(n+1) :=
                        P^(n) · ⟨(q1, q2) | (q1, q2) ∈ Proj2(N) & (q1, q2) ∉ P^(n)⟩;
66              Δ^(n+1) :=
                Δ^(n) ∪ {(n + 1, (a, b), #_{P^(n+1)}((q1, q2))) | ((a, b), (q1, q2)) ∈ N};
67          until n = |P^(n)|
68          ;
69      F :=
            {q | q ∈ {1, ..., |P|} & (Proj1((P)_q) ∈ F1) ∧ (Proj2((P)_q) ∈ F2)};
70      I := {q | q ∈ {1, ..., |P|} & (Proj1((P)_q) ∈ I1) ∧ (Proj2((P)_q) ∈ I2)};
71      ;
```

First, in Lines 51, 52 we define the extended transition relations $E(\Delta_1)$ and $E(\Delta_2)$ (cf. Section 4.2). In Lines 53, 54 the functions Δ'_1 and Δ'_2 are defined that map a state of a source automaton to a set of pairs of label and destination state. The use of these functions helps to avoid a special treatment of each letter of the alphabet Σ. Then, the list of pairs of states P and the resulting transition relation Δ are constructed by induction. In the base step in Lines 58-59, P is set to the list consisting of the pairs in $I_1 \times I_2$, and Δ is empty. In the inductive step we consider (p_1, p_2) – the $n+1$-st element of the list P. Similarly as in Program 8.1.10, the set N defined in Line 62-64 describes the set of all transitions departing from the source (product-) state (p_1, p_2) in a particular way. Here the elements of N are pairs of pairs of the form $((a, b), (q_1, q_2))$, entries a, b are labels and q_1, q_2 are the destinations of the corresponding transitions from p_1 and p_2. In Line 65 we add to P those pairs of destination states in N that are yet not found in P. In Line 66 the function Δ is extended with the transitions from the current pair of states to the corresponding pairs defined with N. For defining Δ all pairs of states are mapped to their corresponding indices in the list P. The induction ends when the list P is exhausted (Line 67). The final states of the product transducer are the indices of all pairs of final

states (Line 69), and the initial states of the product transducer are the indices of pairs of initial states (Line 70). The final automaton is obtained after removing $\langle\varepsilon,\varepsilon\rangle$-transitions and trimming (Line 50).

Program 8.2.4 Following Proposition 4.2.1 the following program presents constructions for **projections** of a 2-tape automaton on the first and second tape, **inverse relation** and the **identity** relation for a given automaton language.

```
72  domain_FST, range_FST : FST → FSA;
73  domain_FST(Σ, Q, I, F, Δ) := (Σ, Q, I, F, {(q, a, r) | (q, (a, b), r) ∈ Δ});
74  range_FST(Σ, Q, I, F, Δ) := (Σ, Q, I, F, {(q, b, r) | (q, (a, b), r) ∈ Δ});
75  inverse_FST : FST → FST;
76  inverse_FST(Σ, Q, I, F, Δ) := (Σ, Q, I, F, {(q, (b, a), r) | (q, (a, b), r) ∈ Δ});
77  identity_FSA : FSA → FST;
78  identity_FSA(Σ, Q, I, F, Δ) := (Σ, Q, I, F, {(q, (a, a), r) | (q, a, r) ∈ Δ});
```

The functions $\text{domain}_{\text{FST}}$ and $\text{range}_{\text{FST}}$ (Lines 72-74) respectively construct the first and second projection of the given finite-state transducer in accordance with Part 2 of Proposition 4.2.1. The function $\text{inverse}_{\text{FST}}$ (Lines 75-76) builds the inverse of a relation (Proposition 4.2.1, Part 3) represented by a finite-state transducer, and the function $\text{identity}_{\text{FSA}}$ (Lines 77-78) constructs the identity relation (Proposition 4.2.1, Part 4) over the language of a finite-state automaton.

Program 8.2.5 The next algorithm, given a finite-state transducer, constructs an equivalent **classical** 2-tape letter automaton, which means that all transition labels are in $\Sigma^{\varepsilon} \times \Sigma^{\varepsilon}$. Following Proposition 4.3.3 we expand the transitions containing labels with words with more than one symbol by introducing new intermediate states.

```
79  expand_FST : FST → FST;
80  expand_FST(Σ, Q, I, F, Δ) := (Σ, Q', I, F, Δ'), where
81      L := ⟨(q', (a'_1, a'_2), r') | (q', (a'_1, a'_2), r') ∈ Δ & (|a'_1| > 1) ∨ (|a'_2| > 1)⟩;
82      (Q', Δ') := induction
83          step 0 :
84              (Q'^(0), Δ'^(0)) := (Q, Δ);
85          step n + 1 :
86              (q, (a_1, a_2), r) := (L)_{n+1};
87              l := max({|a_1|, |a_2|});
88              m := ⟨q⟩ · kNewStates(l − 1, Q'^(n)) · ⟨r⟩;
89              Q'^(n+1) := Q'^(n) ∪ set(m);
90              Δ'^(n+1) := Δ'^(n) \ {(q, (a_1, a_2), r)} ∪
    {((m)_i, (σ(a_1, i), σ(a_2, i)), (m)_{i+1}) | i ∈ {1, ..., l}}, where
91                  σ : WORD × ℕ → WORD;
92                  σ(α, i) := { ⟨(α)_i⟩   if i ≤ |α|
                               { ε         otherwise;
93                  ;
94          until n = |L|
95          ;
96      ;
```

This algorithm closely resembles Program 8.1.7. Here we check if any of the two words in the label of a transition has length > 1 in Line 81. The number of new transitions is determined by the longest label component, l is its length (Line 87). In the new transitions, label components on one side (shorter component of input label) can be empty (Lines 91-92).

We now present the algorithm for composition.

Program 8.2.6 The following algorithm constructs the finite-state transducer that represents the **composition** of two finite-state transducers according to Proposition 4.3.5. Note that in Proposition 4.3.5 we assumed that the two input automata are *letter* automata. Here we ensure this assumption by an explicit conversion to letter automata. In the inductive construction below only states of the target transducer are built that can be reached from an initial state.

97 $\text{compose}_{\text{FST}} : \mathcal{FST} \times \mathcal{FST} \to \mathcal{FST}$;
98 $\text{compose}_{\text{FST}}(A_1, A_2) :=$
$\text{trim}_{\text{FST}}(\text{removeEpsilon}_{\text{FST}}(\Sigma_1 \cup \Sigma_2, \{1, \ldots, |P|\}, I, F, \Delta))$, **where**
99 $(\Sigma_1, Q_1, I_1, F_1, \Delta_1) := \text{expand}_{\text{FST}}(A_1)$;
100 $(\Sigma_2, Q_2, I_2, F_2, \Delta_2) := \text{expand}_{\text{FST}}(A_2)$;
101 $E_{\Delta_1} := \Delta_1 \cup \{(q, (\varepsilon, \varepsilon), q) \mid q \in Q_1\}$;
102 $E_{\Delta_2} := \Delta_2 \cup \{(q, (\varepsilon, \varepsilon), q) \mid q \in Q_2\}$;
103 $\Delta_1' := \mathcal{F}_{1\to(2,3)}(E_{\Delta_1})$;
104 $\Delta_2' := \mathcal{F}_{1\to(2,3)}(E_{\Delta_2})$;
105 $(P, \Delta) \in (\mathcal{STATE} \times \mathcal{STATE})^* \times 2^{\mathcal{TTRANSITION}}$;
106 $(P, \Delta) :=$ **induction**
107 **step** 0 :
108 $P^{(0)} := I_1 \times I_2$ **as** $(\mathcal{STATE} \times \mathcal{STATE})^*$;
109 $\Delta^{(0)} := \emptyset$;
110 **step** $n + 1$:
111 $(p_1, p_2) := (P^{(n)})_{n+1}$;
112 $D_1 := \Delta_1'(p_1)$;
113 $D_2 := \Delta_2'(p_2)$;
114 $N := \{((a_1, b_2), (q_1, q_2)) \mid ((a_1, c), q_1) \in D_1, ((c, b_2), q_2) \in D_2\}$;
115 $P^{(n+1)} := P^{(n)} \cdot \langle p \mid p \in \text{Proj}_2(N)$ & $p \notin P^{(n)} \rangle$;
116 $\Delta^{(n+1)} :=$
$\Delta^{(n)} \cup \{(n + 1, (a_1, b_2), \#_{P^{(n+1)}}(q)) \mid ((a_1, b_2), q) \in N\}$;
117 **until** $n = \left|P^{(n)}\right|$
118 ;
119 $F :=$
$\{q \mid q \in \{1, \ldots, |P|\}$ & $(\text{Proj}_1((P)_q) \in F_1) \wedge (\text{Proj}_2((P)_q) \in F_2)\}$;
120 $I := \{q \mid q \in \{1, \ldots, |P|\}$ & $(\text{Proj}_1((P)_q) \in I_1) \wedge (\text{Proj}_2((P)_q) \in I_2)\}$;
121 ;

The construction is similar to Program 8.2.3. Again we apply the product construction. But first we ensure that the labels in the source transducers are in $\Sigma^\varepsilon \times \Sigma^\varepsilon$ by applying the above $\text{expand}_{\text{FST}}$ function (Lines 99-100). Afterwards we proceed as in Program 8.2.3. As in our earlier product constructions the set N gives a description of all transitions departing from the source pair (p_1, p_2).

In the present case N is defined as the set of pairs of the form $((a_1, b_2), (q_1, q_2))$ for which there exists a string $c \in \Sigma^{\varepsilon}$ such that $(p_1, (a_1, c), q_1) \in E(\Delta_1)$ and $(p_2, (c, b_2), q_2) \in E(\Delta_2)$. Hence (a_1, b_2) is the label of the new transition and (q_1, q_2) is the target pair of states.

Program 8.2.7 The next algorithm constructs the finite-state transducer that represents the **reversal** of a finite-state transducer according to Proposition 4.3.6.

$\mathrm{reverse}_{\mathrm{FST}} : \mathcal{FST} \to \mathcal{FST}$;
$\mathrm{reverse}_{\mathrm{FST}}(\Sigma, Q, I, F, \Delta) := (\Sigma, Q, F, I, \Delta')$, **where**
  $\Delta' := \{(p, (\rho(a), \rho(b)), q) \mid (q, (a, b), p) \in \Delta\}$;
  ;

The above construction directly follows the description given in Proposition 4.3.6.

Real-time translation and pseudo-determinization of transducers

Our next programs describe translation to real-time transducers and pseudo-determinization of transducers.

Program 8.2.8 The following algorithms implement the removal of transducer transitions with label ε on the upper tape as described in the proof of Proposition 4.4.8, and the conversion of a transducer to a **real-time transducer**.

$\mathrm{C}_{\epsilon_{\mathrm{FST}}} : 2^{\mathcal{STATE} \times \mathcal{STATE} \times \mathcal{WORD}} \to 2^{\mathcal{STATE} \times \mathcal{STATE} \times \mathcal{WORD}} \times \mathbb{B}$;
$\mathrm{C}_{\epsilon_{\mathrm{FST}}}(A) := (T, inf)$, **where**
  $A' := \mathcal{F}_{1\to(2,3)}(A)$;
  $(T, inf) :=$ **induction**
    step 0 :
      $T^{(0)} := A$;
      $inf^{(0)} := false$;
    step $n+1$:
      $(a, b, u) := (T^{(n)}$ **as** $(\mathcal{STATE} \times \mathcal{STATE} \times \mathcal{WORD})^*)_{n+1}$;
      $inf^{(n+1)} := (a = b) \wedge (u \neq \varepsilon)$;
      $T^{(n+1)} := \begin{cases} T^{(n)} \cup \{(a, c, u \cdot v) \mid (c, v) \in \mathrm{A}'(b)\} & \textbf{if } !\,\mathrm{A}'(b) \\ T^{(n)} & \textbf{otherwise;} \end{cases}$
    until $(n = |T^{(n)}|) \vee inf^{(n)}$
    ;
  ;
$\mathrm{removeUpperEpsilon}_{\mathrm{FST}} : \mathcal{FST} \to \mathcal{FST} \times 2^{\mathcal{WORD}} \times \mathbb{B}$;
$\mathrm{removeUpperEpsilon}_{\mathrm{FST}}(\Sigma, Q, I, F, \Delta) := ((\Sigma, Q, I, F', \Delta'), W, inf)$, **where**
  $(C_0, inf) := \mathrm{C}_{\epsilon_{\mathrm{FST}}}(\{(q, r, s) \mid (q, (a, s), r) \in \Delta \;\&\; a = \varepsilon\})$;
  $C := C_0 \cup \{(q, q, \varepsilon) \mid q \in Q\}$;
  $W := \{w \mid (q, r, w) \in C \;\&\; (q \in I) \wedge (r \in F)\}$;
  $F' := F \cup \{q \mid (q, r, w) \in C \;\&\; (q \in I) \wedge (r \in F)\}$;
  $C_{\epsilon} := \mathcal{F}_{1\to(2,3)}(C)$;
  $C'_{\epsilon} := \mathcal{F}_{2\to(1,3)}(C)$;

```
148     Δ' := {(q1, (a, u · v · w), q2) | (q', (a, v), q'') ∈ Δ & a ≠ ε,
                                  (q1, u) ∈ C'_ε(q'), (q2, w) ∈ C_ε(q'')};
149     ;
150 realTime_FST : FST → FST × 2^WORD × IB;
151 realTime_FST(A) :=
    removeUpperEpsilon_FST(expand_FST(removeEpsilon_FST(trim_FST(A))));
```

The function $\mathrm{C}_{\epsilon_{\mathrm{FST}}}$ (Lines 126-139) is similar to the function transitiveClosure defined in Program 7.1.3. It computes the transitive closure of the ε-transitions. Here we also concatenate the corresponding outputs on the ε-path in the third projection of T (Line 136). A second distinction to Program 7.1.3 is that the current program in addition tests for an ε-loop with non-empty output (Line 135). The program is used for a trimmed input transducer - if such a loop occurs, then the transducer is infinitely ambiguous (cf. Proposition 4.5.7). In this case the induction terminates (Line 137), which is indicated by the resulting flag *inf*.

The function $\mathrm{removeUpperEpsilon}_{\mathrm{FST}}$ (Lines 140-149) then first constructs the reflexive-transitive ε-closure in Line 142. In Line 144 we define W as the set of all possible outputs for input ε. For functional transducers the set W will have not more than one element. To the final states we add the initial states from which final states are ε-reachable (Line 145). In Lines 146-147 the ε-closure and its inverse relation are functionalized (cf. Definition 1.1.16). Finally in Line 148 Δ' is defined as in Proposition 4.4.8. The function returns the resulting transducer together with the set W of outputs for ε.

The function $\mathrm{realTime}_{\mathrm{FST}}$ (Lines 150-151) first trims the input transducer and removes the $\langle\varepsilon,\varepsilon\rangle$ transitions, then expands the transitions with labels having more than one symbol (cf. Program 8.2.5) and finally removes the transitions with label ε on the upper tape.

Program 8.2.9 The next algorithm constructs a **pseudo-deterministic transducer** equivalent to a given input transducer using the steps described in the proof of Proposition 3.7.2.

```
152 pseudoDeterm_FST : FST → FST;
153 pseudoDeterm_FST(Σ, Q, I, F, Δ) :=
    trim_FST(Σ, {1, ..., |P|}, {1}, F', δ'), where
154     Δ' := F_{1→(2,3)}(Δ);
155     (P, δ') ∈ (2^STATES)* × 2^TTRANSITION;
156     (P, δ') := induction
157         step 0 :
158             P^(0) := ⟨I⟩;
159             δ'^(0) := ∅;
160         step n + 1 :
161             N := ⋃(Δ'((P^(n))_{n+1}));
162             N' := F_{1→2}(N);
163             P^(n+1) := P^(n) · ⟨q | q ∈ Proj_2(N') & q ∉ P^(n)⟩;
164             δ'^(n+1) := δ'^(n) ∪ {(n + 1, c, #_{P^(n+1)}(q)) | (c, q) ∈ N'};
165         until n = |P^(n)|
166         ;
167     F' := {q | q ∈ {1, ..., |P|} & F ∩ (P)_q ≠ ∅};
```

168 ;

The algorithm can be considered as a variant or extension of the determinization Program 8.1.9. We proceed with the determinization in the same way as in Program 8.1.9. For Line 154, recall that Δ is a ternary relation. We treat the transition labels, which are pairs of words as individual symbols. Finally, the function returns the pseudo-deterministic transducer after trimming.

Pseudo-minimization of transducers

Program 8.2.10 The following algorithm constructs a **pseudo-minimal transducer** following Proposition 3.7.4.

169 $\text{pseudoMinimal}_{\text{FST}} : \mathcal{FST} \to \mathcal{FST}$;
170 $\text{pseudoMinimal}_{\text{FST}}(A) := \text{trim}_{\text{FST}}(\Sigma, \text{R}(Q), \text{R}(I), \text{R}(F), \delta')$, **where**
171 $(\Sigma, Q, I, F, \Delta) := \text{pseudoDeterm}_{\text{FST}}(A)$;
172 $\Sigma' := \text{Proj}_2(\Delta)$;
173 $\delta : \mathcal{STATE} \times (\mathcal{WORD} \times \mathcal{WORD}) \to \mathcal{STATE}$;
174 $\delta := \{((q,(a,b)),r) \mid (q,(a,b),r) \in \Delta\}$;
175 $R \in \mathcal{EQREL}$;
176 $k \in \mathbb{N}$;
177 $(R,k) :=$ **induction**
178 **step** 0 :
179 $\text{f} : \mathcal{STATE} \to \mathbb{N}$;
180 $\text{f}(q) := \begin{cases} 1 & \textbf{if } q \in F \\ 0 & \textbf{otherwise;} \end{cases}$
181 $k^{(0)} := 0$;
182 $R^{(0)} := \ker(Q, f)$;
183 **step** $n+1$:
184 $\text{f} : (\mathcal{WORD} \times \mathcal{WORD}) \times \mathcal{STATE} \to \mathbb{N}$;
185 $\text{f}(c,q) := \begin{cases} \text{R}^{(\text{n})}(\delta(q,c)) & \textbf{if } !\,\delta(q,c) \\ 0 & \textbf{otherwise;} \end{cases}$
186 $k^{(n+1)} := \left|\text{Proj}_2(R^{(n)})\right|$;
187 $R^{(n+1)} :=$
$\text{intersect}_{\text{EQREL}}(Q, R^{(n)}, (\text{intersect}_{\text{EQREL}}(Q))(\langle \ker(Q, \text{f}(c)) \mid c \in \Sigma' \rangle))$;
188 **until** $(\Sigma' = \emptyset) \vee (k^{(n)} = \left|\text{Proj}_2(R^{(n)})\right|)$
189 ;
190 $\delta' := \{(\text{R}(q), \sigma, \text{R}(r)) \mid (q,\sigma,r) \in \Delta\}$;
191 ;

In Line 171 we first construct the pseudo-deterministic transducer for the given input transducer. Afterwards, the algorithm proceeds with the minimization exactly as in Program 8.1.14. Note that the minimization is performed over the alphabet Σ' consisting of the transition labels (Line 172).

Deciding functionality of transducers

Program 8.2.11 The next program constructs the **squared output transducer** for a real-time transducer in accordance with Definition 4.6.4.

192 $\text{squaredOutputTransducer}_{\text{FST}} : \mathcal{FST} \to \mathcal{FST}$;
193 $\text{squaredOutputTransducer}_{\text{FST}}(\Sigma, Q, I, F, \Delta) :=$
$(\Sigma, \{1, \ldots, |P|\}, I', F', \Delta')$, **where**
194 $\Delta_0 := \mathcal{F}_{1\to(2,3)}(\Delta)$;
195 $(P, \Delta') \in (\mathcal{STATE} \times \mathcal{STATE})^* \times 2^{\mathcal{TTRANSITION}}$;
196 $(P, \Delta') :=$ **induction**
197 **step** 0 :
198 $P^{(0)} := I \times I$ **as** $(\mathcal{STATE} \times \mathcal{STATE})^*$;
199 $\Delta'^{(0)} := \emptyset$;
200 **step** $n+1$:
201 $(p_1, p_2) := (P^{(n)})_{n+1}$;
202 $D_1 := \begin{cases} \Delta_0(p_1) & \textbf{if } !\,\Delta_0(p_1) \\ \emptyset & \textbf{otherwise;} \end{cases}$
203 $D_2 := \begin{cases} \Delta_0(p_2) & \textbf{if } !\,\Delta_0(p_2) \\ \emptyset & \textbf{otherwise;} \end{cases}$
204 $N :=$
$\{((\alpha_1, \alpha_2), (q_1, q_2)) \mid ((a, \alpha_1), q_1) \in D_1, ((a, \alpha_2), q_2) \in D_2\}$;
205 $P^{(n+1)} := P^{(n)} \cdot \langle p \mid p \in \text{Proj}_2(N) \;\&\; p \notin P^{(n)} \rangle$;
206 $\Delta'^{(n+1)} :=$
$\Delta'^{(n)} \cup \{(n+1, (a_1, b_2), \#_{P^{(n+1)}}(q)) \mid ((a_1, b_2), q) \in N\}$;
207 **until** $n = \left|P^{(n)}\right|$
208 ;
209 $F' :=$
$\{q \mid q \in \{1, \ldots, |P|\} \;\&\; (\text{Proj}_1((P)_q) \in F) \wedge (\text{Proj}_2((P)_q) \in F)\}$;
210 $I' := \{q \mid q \in \{1, \ldots, |P|\} \;\&\; (\text{Proj}_1((P)_q) \in I) \wedge (\text{Proj}_2((P)_q) \in I)\}$;
211 ;

The program proceeds in a similar way as Programs 8.2.3 (Cartesian product) and 8.2.6 (composition). Here we create the product of the transducer with itself. Transition labels are defined as in Definition 4.6.4 (Line 204).

Program 8.2.12 This program decides the **functionality** of a transducer following Corollary 4.6.7, Proposition 4.6.8 and Corollary 4.6.11.

212 $\text{commonPrefix}, \text{remainderSuffix} : \mathcal{WORD} \times \mathcal{WORD} \to \mathcal{WORD}$;
213 $\text{commonPrefix}(s', s'') := (s')_{1,\ldots,l}$, **where**
214 $l := \min_{l' \in \{0,\ldots,|s'|\} \;\&\; (l'+1>|s'|) \vee (l'+1>|s''|) \vee ((s')_{l'+1} \neq (s'')_{l'+1})} l'$;
215 ;
216 $\text{remainderSuffix}(w, s) := (s)_{|w|+1,\ldots,|s|}$;
217 $\omega : (\mathcal{WORD} \times \mathcal{WORD}) \times (\mathcal{WORD} \times \mathcal{WORD}) \to \mathcal{WORD} \times \mathcal{WORD}$;
218 $\omega((u, v), (\alpha, \beta)) := (\text{remainderSuffix}(c, u \cdot \alpha), \text{remainderSuffix}(c, v \cdot \beta))$,
where
219 $c := \text{commonPrefix}(u \cdot \alpha, v \cdot \beta)$;
220 ;

```
221  balancible : 𝒲𝒪ℛ𝒟 × 𝒲𝒪ℛ𝒟 → 𝔹;
222  balancible(u, v) := (u = ε) ∨ (v = ε);
223  testFunctionality_FST : ℱ𝒮𝒯 → 𝔹;
224  testFunctionality_FST(T) := ¬inf ∧ (|W| ≤ 1) ∧ functional, where
225      (T′, W, inf) := realTime_FST(T);
226      (Σ, Q, I, F, Δ) := trim_FST(squaredOutputTransducer_FST(T′));
227      Δ_0 := ℱ_{1→(2,3)}(Δ);
228      Adm : 𝒮𝒯𝒜𝒯ℰ → 𝒲𝒪ℛ𝒟 × 𝒲𝒪ℛ𝒟;
229      functional ∈ 𝔹;
230      (Adm, functional) := induction
231          step 0 :
232              Adm^(0) := I × {(ε, ε)};
233              functional^(0) := true;
234          step i + 1 :
235              (q, h) := (Adm^(i) as (𝒮𝒯𝒜𝒯ℰ × (𝒲𝒪ℛ𝒟 × 𝒲𝒪ℛ𝒟))*)_{i+1};
236              D_q := { {(q′, ω(h, U)) | (U, q′) ∈ Δ_0(q)}   if ! Δ_0(q)
                       { ∅                                   otherwise;
237              functional^(i+1) := ∀(q′, h′) ∈ D_q : (balancible(h′) ∧
         ((q′ ∈ F) → (h′ = (ε, ε))) ∧ (! Adm^(i)(q′) → (h′ = Adm^(i)(q′))));
238              Adm^(i+1) :=
                 { Adm^(i)                                   if ¬functional^(i+1)
                 { Adm^(i) ∪ {(q′, h′) | (q′, h′) ∈ D_q}     otherwise;
239          until ¬functional^(i) ∨ (i = |Adm^(i)|)
240          ;
241      ;
```

The program starts with the definition of the functions commonPrefix ($s' \wedge s''$) in Lines 212-215, remainderSuffix ($w^{-1}s$) in Line 216, advance function ω in Lines 217-220, and the function *balancible*, which checks whether an advance is balancible in Lines 221-222. After this preparation the main function $\text{testFunctionality}_{\text{FST}}$ is introduced. First we construct the trimmed squared output transducer of the corresponding real-time transducer in Lines 225-226. Then we compute the admissible sets for the states of the trimmed squared output transducer using an induction following Corollary 4.6.11 (Lines 230-240). Since the admissible sets of a functional transducer are singletons we define Adm as a function (Line 228). The flag *functional* is defined during the induction for constructing Adm. The flag is returned as a result and shows whether the transducer is functional. In Lines 232-233 we set the base of the induction – the advances of the initial states to $\langle \varepsilon, \varepsilon \rangle$ and the flag *functional* to *true*. Then the induction proceeds with the elements of the function Adm regarded as a list (Line 235). The successors of the considered state are computed in Line 236. Line 237 checks whether the advance h' is balancible for each successor of the considered state, whether h' is equal to $\langle \varepsilon, \varepsilon \rangle$ for the final states, and whether the definition is consistent with the advances of the states defined so far. Line 238 adds the new advances to Adm. The induction terminates when the flag is false or when the elements in Adm are exhausted.

Spell checking as an application

We finish this section with an example demonstrating how to make use of the presented programs for realizing interesting applications.

Example 8.2.13 The following program realizes the functionality of a fully-fledged spell checker. It implements a function testing for an input word w if w is in a given background dictionary, also retrieving the set of dictionary words that are "close" to the given word. This set represents the correction candidates for the given word. Closeness is defined in terms of the Levenshtein distance. We recall that the Levenshtein distance between two words is the minimal number of symbol substitutions, deletions and insertions required for transforming the first word into the second one. Two short words (up to 5 letters) are considered to be close if their distance is one. Two medium words (between 6 and 10 symbols) are close if their distance is less or equal to two. Two long words (11 or more symbols) are close if their distance is less or equal to three. Below the code is presented. This code can be applied on any suitable word list[1].

```
1  import ← "Section_8.2.cm";
2  readWordList : 𝒲𝒪ℛ𝒟 → ℱ𝒮𝒜;
3  readWordList(FileName) := D, where
4      F := loadText(FileName);
5      p ∈ ℕ;
6      (D, p) := induction
7          step 0 :
8              (D^(0), p^(0)) := ((∅, ∅, ∅, ∅, ∅), 0);
9          step k + 1 :
10             p' ∈ ℕ;
11             p' := induction
12                 step 0 :
13                     p'^(0) := p^(k) + 1;
14                 step l + 1 :
15                     p'^(l+1) := p'^(l) + 1;
16                 until (p'^(l) > |F|) ∨ ((F)_{p'^(l)} =' \n')
17                 ;
18             D^(k+1) := union_FSA(D^(k), FSA_WORD((F)_{p^(k)+1,...,p'-1}));
19             p^(k+1) := p';
20         until p^(k) ≥ |F|
21         ;
22     ;
```

We start with an auxiliary function for reading a word list from a file and constructing a finite-state automaton representing the set of words in the word list. We assume that the word list contains one word per line. After importing the programs from the previous section (Line 1) we define the function readWordList, which has as parameter the file name of the word list and returns the constructed automaton (Lines 2-3). In Line 4 the content of the file is read

[1]Spell checker oriented word lists can be obtained from http://wordlist.aspell.net

in the text F using the build-in function loadText (Line 4). Afterwards, in the induction in Lines 6-21 we run through the text F and add each word to the resulting automaton D. In p we maintain the position of the ending of the last processed word in F. The induction is initialized in Line 8 with the empty automaton and the zero index. In the inductive step we first find the index p' of the end of the next word by the sub-induction in Lines 11-17. We simply test each consecutive symbol in F for new line or end of file (Line 16). Then in Line 18 we add to the automaton D the next word, which appears in the text between positions $p^{(k)} + 1$ and $p' - 1$. Finally the position $p^{(k)}$ is updated in Line 19. The induction ends when the text F is exhausted (Line 20).

23 $\mathrm{L_A} : \mathcal{DFSA} \to 2^{\mathcal{WORD}}$;
24 $\mathrm{L_A}(\Sigma, Q, q_0, F, \delta) := L$, **where**
25 $P \in 2^{\mathcal{STATE} \times \mathcal{WORD}}$;
26 $(P, L) :=$ **induction**
27 **step** 0 :
28 $P^{(0)} := \{(q_0, \varepsilon)\}$;
29 $L^{(0)} := \begin{cases} \{\varepsilon\} & \textbf{if } q_0 \in F \\ \emptyset & \textbf{otherwise} \end{cases}$;
30 **step** $i + 1$:
31 $P^{(i+1)} := \{(\delta(q, a), w \cdot \langle a \rangle) \mid (q, w) \in P^{(i)}, a \in \Sigma \;\&\; !\,\delta(q, a)\}$;
32 $L^{(i+1)} := L^{(i)} \cup \{w \mid (q, w) \in P^{(i+1)} \;\&\; q \in F\}$;
33 **until** $\left|P^{(i)}\right| = 0$
34 ;
35 ;

The function L_A presented in Lines 23-35 returns the language of an acyclic deterministic finite-state automaton. We obtain the language L proceeding by a simple induction. At the i-th step we maintain in $P^{(i)}$ the set of pairs consisting of a state reachable with a word of length i and the corresponding word. In the initialisation step we initialize $P^{(0)}$ with the initial state q_0 and the empty word in Line 28. $L^{(0)}$ is the singleton set with the empty word if $q_0 \in F$, otherwise it is empty. Then in the inductive step we obtain $P^{(i+1)}$ from $P^{(i)}$ in Line 31 and extend L in Line 32. The induction ends when $P^{(i)}$ gets empty (Line 33).

36 $Dic := \mathrm{readWordList}(\text{"en\_US.dic"})$;
37 $Dic_{DFSA} := \mathrm{minimal_{DFSA}}(\mathrm{determ_{FSA}}(Dic))$;
38 $Symbol := \mathrm{symbolSet2FSA}(\mathrm{Proj_1}(Dic_{DFSA}))$;
39 $ID := \mathrm{star_{FST}}(\mathrm{identity_{FSA}}(Symbol))$;
40 $Del := \mathrm{product_{FSA}}(Symbol, \mathrm{FSA_{WORD}}(\text{""}))$;
41 $Ins := \mathrm{product_{FSA}}(\mathrm{FSA_{WORD}}(\text{""}), Symbol)$;
42 $Sub := \mathrm{product_{FSA}}(Symbol, Symbol)$;
43 $Lev1 := \mathrm{concat_{FST}}(\langle ID, \mathrm{option_{FST}}(\mathrm{union_{FST}}(\langle Del, Ins, Sub \rangle)), ID \rangle)$;
44 $Lev2 := \mathrm{concat_{FST}}(Lev1, Lev1)$;
45 $Lev3 := \mathrm{concat_{FST}}(Lev2, Lev1)$;
46 $\mathrm{spellCheck} : \mathcal{WORD} \to \mathbb{B} \times 2^{\mathcal{WORD}}$;
47 $\mathrm{spellCheck}(w) := (inDic, Corrections)$, **where**
48 $(\Sigma_D, Q_D, q_0, F_D, \delta_D) := Dic_{DFSA}$;
49 $\pi := \mathrm{stateseq}(\delta_D, q_0, w)$;
50 $inDic := (|\pi| = |w| + 1) \wedge ((\pi)_{|\pi|} \in F_D)$;

51 $Lev := \begin{cases} Lev1 & \textbf{if } |w| < 6 \\ Lev2 & \textbf{if } |w| < 11 \\ Lev3 & \textbf{otherwise} \end{cases}$;

52 $LevensteinAutomaton := \text{determ}_{\text{FSA}}(\text{range}_{\text{FST}}(\text{compose}_{\text{FST}}(\text{identity}_{\text{FSA}}(\text{FSA}_{\text{WORD}}(w)), Lev)))$;

53 $Corrections := \text{L}_{\text{A}}(\text{intersect}_{\text{DFSA}}(Dic_{DFSA}, LevensteinAutomaton))$;

54 ;

After the preparation we proceed with the implementation of the spell checker. First, we construct the minimal deterministic finite-state automaton for the dictionary in Lines 36-37. In Lines 38-42 we respectively construct the transducers representing single symbol deletions, insertions, and substitutions. Line 43 constructs the transducer that represents the relation mapping a word to any word in distance up to 1. By simple concatenation the corresponding transducers representing distance up to 2 and 3 are built in Lines 44-45. The function spellCheck defined in Lines 46-54 returns for the given word w a flag *inDic* for the presence of w in the dictionary and the set of correction candidates *Corrections*. In Line 49 the states on the path with input w in the dictionary automaton is obtained. If the list has $|w| + 1$ states and the last state on the path is final, then w is in the dictionary (Line 50). Afterwards an appropriate Levenshtein filter depending on the length of w is chosen in Line 51. The Levenshtein automaton for the word w is defined as the deterministic finite-state automaton representing all words that are close to w (Line 52). The correction candidates are obtained by intersecting the Levenshtein automaton for w with the dictionary automaton in Line 53.

We applied the above program using a standard English word list containing 48.000 entries. The dictionary automaton has approximately 28.500 states and 60.000 transitions.

Remark 8.2.14 The above functions can be applied for large dictionaries and provide a practical solution to spell checking. Nevertheless some steps can be substantially improved.

1. The minimal deterministic finite-state automaton for the dictionary can be constructed in a much more efficient way. This will be shown in Chapter 10.
2. The deterministic Levenshtein automaton can be constructed directly using the method in [Schulz and Mihov, 2002].
3. The words in the intersection of the dictionary automaton and the Levenshtein automaton can be obtain more efficiently with a parallel traversal procedure, avoiding the construction of the intersection automaton.

Remark 8.2.15 Another method for retrieving the set of correction candidates is the following. We can directly construct the transducer for mapping each dictionary word w to the words that are close to w. By inverting this relation we obtain a transducer which maps each word w to the dictionary words close to w. We can retrieve correction candidates by traversing this transducer. The drawback of this method is the size of the transducer. For the English word list it has more than 30 million transitions.

8.3 *C(M)* programs for deterministic transducers

In this section we present the algorithms for construction and minimization of deterministic transducers. We assume that all functions defined in Sections 8.1 and 8.2 are available.

Program 8.3.1 The following algorithm constructs a **subsequential finite-state transducer** from a finite-state transducer with the bounded variation property, closely following the inductive construction presented in Section 5.2. We generalize the construction at one minor point, tolerating situations where the empty input is translated to a single non-empty word (Lines 36-40).

```
1  import ← "Section_8.2.cm";
2  TRANSITIONOUT is STATE × SYMBOL → WORD;
3  STATEOUT is STATE → WORD;
4  SSFST is ALPHABET × 2^STATE × STATE × 2^STATE ×
   DTRANSITIONS × TRANSITIONOUT × STATEOUT;
5  ssfst_FST : FST → SSFST × 𝔹;
6  ssfst_FST(A) := (T, boundedVariation), where
7      ((Σ, Q, I, F, Δ), W, inf) := realTime_FST(A);
8      C := max_{(p,(a,α),q)∈Δ} |α| ;
9      Z := C × |Q|^2 ;
10     Δ' := F_{1→(2,3)}(Δ);
11     P ∈ 2^{2^{STATE×WORD}} ;
12     δ' ∈ DTRANSITIONS;
13     λ' ∈ TRANSITIONOUT;
14     Ψ' ∈ STATEOUT;
15     (P, δ', λ', boundedVariation) := induction
16         step 0 :
17             P^(0) := {I × {ε}};
18             δ'^(0) := ∅;
19             λ'^(0) := ∅;
20             boundedVariation^(0) := true;
21         step n + 1 :
22             S := (P^(n) as (2^{STATE×WORD})*)_{n+1};
23             N := {(σ, u, v, q') | (q, u) ∈ S & ! Δ'(q), ((σ, v), q') ∈ Δ'(q)};
24             N' := F_{1→(2,3,4)}(N);
25             N'' := {(σ, w, S') | (σ, X) ∈ N',
                       w = commonPrefix(⟨u · v | (u, v, q') ∈ X⟩),
                       S' = {(q', remainderSuffix(w, u · v)) | (u, v, q') ∈ X}};
26             boundedVariation^(n+1) := ∀(σ, w, S') ∈ N'', (q, u) ∈ S' : (|u| < Z);
27             P^(n+1) := P^(n) ∪ {S' | (σ, w, S') ∈ N''};
28             δ'^(n+1) :=
   δ'^(n) ∪ {((n + 1, (σ)_1), #_{P^(n+1)} as (2^{STATE×WORD})*(S')) | (σ, w, S') ∈ N''};
29             λ'^(n+1) := λ'^(n) ∪ {((n + 1, (σ)_1), w) | (σ, w, S') ∈ N''};
30         until ¬boundedVariation^(n) ∨ (n = |P^(n)|)
31         ;
```

```
32      F' :=
    {s | s ∈ {1,...,|P|}, S = (P as (2^{STATE×WORD})*)_s & ∃(q,β) ∈ S : (q ∈ F)};
33      Ψ' := {(s,β) | s ∈ {1,...,|P|}, S = (P as (2^{STATE×WORD})*)_s &
                ∃(q',β') ∈ S : (q' ∈ F), β = elementOf({β' | (q',β') ∈ S & q' ∈ F})};
34      T :=
35          case (W = ∅) ∨ (W = {ε}) : (Σ, {1,...,|P|}, 1, F', δ', λ', Ψ')
36          otherwise :
    (Σ, {1,...,|P|+1}, q'_0, F' ∪ {q'_0}, δ'', λ'', Ψ' ∪ {(q'_0, elementOf(W))}), where
37              q'_0 := |P| + 1;
38              δ'' := δ' ∪ {((q'_0, σ), δ'(1,σ)) | σ ∈ Σ & !δ'(1,σ)};
39              λ'' := λ' ∪ {((q'_0, σ), λ'(1,σ)) | σ ∈ Σ & !δ'(1,σ)};
40          ;
41      ;
```

After importing the programs from the previous sections in Line 1 in Lines 2-4 we define the types needed for subsequential transducers. In Line 7 we first construct a real-time finite-state transducer equivalent to the input transducer A. Afterwards we proceed with the determinization in a similar way as in Program 8.1.9. The states P, the transition function δ', the transition output function λ', and the state output function Ψ' are defined exactly as described in the inductive construction given in Section 5.2. The notation $(\sigma)_1$ shows the extraction of the first (and only) letter from the string σ. During the induction we test for violation of the bounded variation property in accordance with Remark 5.3.9 in Line 26. The induction is completed when violation of the bounded variation property is encountered or when all states are processed (Line 30). The 'elementOf'-operator returns an element of the set used as argument. Finally, if there is no output for the empty input, or if the output for the empty input is the empty word (Line 35), then we are done. Otherwise we clone the starting state and add a new starting state with the desired output for the empty input (Lines 36-40).

Deciding bounded variation of transducers

Program 8.3.2 The next program tests the **bounded variation poperty** for a transducer in accordance with Theorem 5.3.4 and Lemma 5.3.5.

```
42  testBoundedVariation_FST : FST → IB;
43  testBoundedVariation_FST(T) := boundedVariation, where
44      ((Σ', Q', I', F', Δ'), W, inf) := realTime_FST(T);
45      C := max_{(p,(a,α),q)∈Δ'} |α| ;
46      Z := C × |Q'|^2 ;
47      (Σ, Q, I, F, Δ) := squaredOutputTransducer_FST(Σ', Q', I', F', Δ');
48      Δ_0 := F_{1→(2,3)}(Δ);
49      Adm ∈ 2^{STATE×(WORD×WORD)};
50      boundedVariation ∈ IB;
51      (Adm, boundedVariation) := induction
52          step 0 :
53              Adm^(0) := I × {(ε,ε)};
54              boundedVariation^(0) := true;
```

55 **step** $i+1$:
56 $(q,h) := (Adm^{(i)}$ **as** $(\mathcal{STATE} \times (\mathcal{WORD} \times \mathcal{WORD}))^*)_{i+1}$;
57 $D_q := \begin{cases} \{(q', \omega(h,U)) \mid (U,q') \in \Delta_0(q)\} & \textbf{if } !\,\Delta_0(q) \\ \emptyset & \textbf{otherwise;} \end{cases}$
58 $boundedVariation^{(i+1)} :=$
$\forall (q',(u',v')) \in D_q : ((|u'| < Z) \wedge (|v'| < Z))$;
59 $Adm^{(i+1)} := \begin{cases} Adm^{(i)} & \textbf{if } \neg boundedVariation^{(i+1)} \\ Adm^{(i)} \cup D_q & \textbf{otherwise;} \end{cases}$
60 **until** $\neg boundedVariation^{(i)} \vee (i = \left|Adm^{(i)}\right|)$
61 ;
62 ;

The above program proceeds similarly as Program 8.2.12. In Lines 45-46 the upper limit derived in Lemma 5.3.5 is calculated. Then we proceed by first obtaining the (non-trimmed) squared output transducer (Line 47), and inductively calculating the admissible advances of its states afterwards (Lines 48-60). In contrast to Program 8.2.12, the admissible advances are a relation (Line 49) and the flag *boundedVariation* checks whether the induction leads to an infinite number of states by testing the sizes of the words in the advances (Lemma 5.3.5) in Line 58. The flag *boundedVariation* is returned as result.

Minimization of subsequential transducers

The algorithms for the minimization of subsequential finite-state transducers are presented below.

Program 8.3.3 The program below defines the type for **subsequential finite-state transducers with initial output** and the conversion procedure from and to ordinary subsequential finite-state transducer according to Definition 5.4.4 and Proposition 5.4.5.

63 $\mathcal{SSFSTI}$ **is** $\mathcal{ALPHABET} \times 2^{\mathcal{STATE}} \times \mathcal{STATE} \times 2^{\mathcal{STATE}} \times$
$\mathcal{DTRANSITIONS} \times \mathcal{TRANSITIONOUT} \times \mathcal{WORD} \times \mathcal{STATEOUT}$;
64 $\text{SSFSTI}_{\text{SSFST}} : \mathcal{SSFST} \to \mathcal{SSFSTI}$;
65 $\text{SSFSTI}_{\text{SSFST}}(\Sigma, Q, q_0, F, \delta, \lambda, \Psi) := (\Sigma, Q, q_0, F, \delta, \lambda, \varepsilon, \Psi)$;
66 $\text{SSFST}_{\text{SSFSTI}} : \mathcal{SSFSTI} \to \mathcal{SSFST}$;
67 $\text{SSFST}_{\text{SSFSTI}}(\Sigma, Q, q_0, F, \delta, \lambda, \iota, \Psi) :=$
68 **case** $\iota = \varepsilon : (\Sigma, Q, q_0, F, \delta, \lambda, \Psi)$
69 **case** $\exists((q', \sigma), q'') \in \delta : (q'' = q_0) : (\Sigma, Q \cup \{q_0'\}, q_0', F', \delta', \lambda', \Psi')$, **where**
70 $q_0' := |Q| + 1$;
71 $F' := F \cup \begin{cases} \{q_0'\} & \textbf{if } q_0 \in F \\ \emptyset & \textbf{otherwise;} \end{cases}$
72 $\Psi' := \Psi \cup \begin{cases} \{(q_0', \iota \cdot \Psi(q_0))\} & \textbf{if } q_0 \in F \\ \emptyset & \textbf{otherwise;} \end{cases}$
73 $\delta' := \delta \cup \{((q_0', \sigma), \delta(q_0, \sigma)) \mid \sigma \in \Sigma \;\&\; !\,\delta(q_0, \sigma)\}$;
74 $\lambda' := \lambda \cup \{((q_0', \sigma), \iota \cdot \lambda(q_0, \sigma)) \mid \sigma \in \Sigma \;\&\; !\,\delta(q_0, \sigma)\}$;
75 **otherwise** $: (\Sigma, Q, q_0, F, \delta, \lambda', \Psi')$, **where**
76 $\Psi' :=$

```
77              case q_0 ∈ F : Ψ \ {(q_0, o_0)} ∪ {(q_0, ι · o_0)}, where
78                  o_0 := Ψ(q_0);
79              otherwise : Ψ
80              ;
81          λ_0 ∈ TRANSITIONOUT;
82          λ_0 := {((q_0, σ), λ(q_0, σ)) | σ ∈ Σ & !δ(q_0, σ)};
83          λ' := λ \ λ_0 ∪ {((q'_0, σ), ι · l_0) | ((q'_0, σ), l_0) ∈ λ_0};
84      ;
```

Line 63 defines the type for subsequential finite-state transducer with initial output. Lines 64-65 present the conversion from ordinary subsequential finite-state transducers to subsequential finite-state transducers with initial output by setting ι to the empty word. For the reverse conversion three cases are distinguished. If $\iota = \varepsilon$, we just abandon the initial output ι (Line 68). If the starting state is reachable, then we introduce a new starting state q'_0 and follow the procedure from Proposition 5.4.5 (Lines 69-74). In the third case we just add the initial output ι in front of all outputs from the starting state (Lines 75-83).

Program 8.3.4 This algorithm returns the **expanded output automaton** for a given subsequential transducer as defined in Definition 5.5.11. Functions from Program 8.1.5 and Program 8.1.7 are used.

```
85  expandedOutputAutomaton : SSFSTI → FSA;
86  expandedOutputAutomaton(T) :=
    expand_FSA(removeEpsilonPreserving_FSA(A_T)), where
87      A_T := (Σ, Q ∪ {f}, {q_0}, {f}, Δ), where
88          (Σ, Q, q_0, F, δ, λ, ι, Ψ) := T;
89          f := |Q| + 1;
90          Δ := {(q, Ψ(q), f) | q ∈ F} ∪ {(q', λ(q', σ), q'') | ((q', σ), q'') ∈ δ};
91          ;
92      ;
```

Program 8.3.5 The function below calculates the **maximal state output function** *mso* for a given transducer $\mathcal{T}$ according to Corollary 5.5.15.

```
93  mso_SSFSTI : SSFSTI → STATE → WORD;
94  mso_SSFSTI(T) := mso, where
95      Q_T := Proj_2(T);
96      (Σ, Q, I, F, Δ) := expandedOutputAutomaton(T);
97      Δ' := F_{1→(2,3)}({(q, (α)_1, r) | (q, α, r) ∈ Δ});
98      R ∈ (2^STATE)*;
99      δ' : STATE → SYMBOL × STATE;
100     (R, δ') := induction
101         step 0 :
102             R^(0) := ⟨{q} | q ∈ Q_T⟩;
103             δ'^(0) := ∅;
104         step n + 1 :
105             N := ⋃(Δ'((R^(n))_{n+1}));
106             N' := F_{1→2}(N);
```

```
107             (R^(n+1), δ'^(n+1)) :=
108                 case ((R^(n))_{n+1} ∩ F ≠ ∅) ∨ (|N'| ≠ 1) : (R^(n), δ'^(n))
109                 otherwise :
                (R', δ'^(n) ∪ {(n + 1, (c, #_{R'}(q))) | (c, q) ∈ N'}), where
110                     R' := R^(n) · ⟨q | q ∈ Proj_2(N') & q ∉ R^(n)⟩;
111                 ;
112         until n = |R^(n)|
113         ;
114     LCP_q : STATE → WORD;
115     LCP_q(q) := w, where
116         q' ∈ STATE;
117         (q', w) := induction
118             step 0 :
119                 w^(0) := ε;
120                 q'^(0) := q;
121             step i + 1 :
122                 (c, q'^(i+1)) := δ'(q'^(i));
123                 w^(i+1) := w^(i) · ⟨c⟩;
124             until ¬(! δ'(q'^(i)))
125             ;
126         ;
127     mso := {(q, LCP_q(q)) | q ∈ Q_T};
128     ;
```

The set of transducer states Q_T is defined in Line 95, and the expanded output automaton for T is obtained in Line 96. Afterwards a special partial determinization procedure is applied (cf. Remark 5.5.16). It differs from the standard determinization procedure in two aspects: (i) it starts from any singleton subset with an original transducer state (Line 102), and (ii) determinization is performed until either a final state is reached or there are more than one outgoing transitions from the state (Line 108). In the program, R describes the sequence of new states obtained, each new state is a set of "old" transducer states. At the beginning the sequence R contains all singleton sets (Line 102). When treating a new state $(R^{(n)})_{n+1}$ we build the set N of all pairs (σ, q') such that $(q, \sigma, q') \in \Delta$ for some q in $(R^{(n)})_{n+1}$. The function N' maps each relevant transition symbol σ to the new state (sets of old states) reached with σ from $(R^{(n)})_{n+1}$. If N' contains more than one pair, then there are transitions with two distinct symbols from $(R^{(n)})_{n+1}$. In that case we stop (Line 108). We also stop if $(R^{(n)})_{n+1}$ contains a final state. Since in the resulting automaton for each state there will be not more than one outgoing transition, we can encode the transitions as a (partial) function δ' that maps a state to a pair of label and destination state (Lines 99, 109). Except these differences the order of determinization steps in Lines 105, 110-113 proceeds similarly as in Program 8.1.9. After the determinization the function $\mathrm{LCP_q}$ (Lines 114-126) follows the path from a given state q and returns the corresponding label. The resulting function *mso* is defined in Line 127.

Program 8.3.6 The following program converts a subsequential transducer into **canonical form** as defined in Definition 5.5.2.

```
129 canonical_SSFSTI : SSFSTI → SSFSTI;
130 canonical_SSFSTI(Σ, Q, q_0, F, δ, λ, ι, Ψ) := (Σ, Q, q_0, F, δ, λ', ι', Ψ'), where
131     mso := mso_SSFSTI(Σ, Q, q_0, F, δ, λ, ι, Ψ);
132     ι' := ι · mso(q_0);
133     Ψ' := {(q, remainderSuffix(mso(q), Ψ(q))) | q ∈ F};
134     λ' :=
            {((q, σ), remainderSuffix(mso(q), l · mso(δ(q, σ)))) | ((q, σ), l) ∈ λ};
135     ;
```

Program 8.3.7 The next program presents the **pseudo-minimization** and **minimization** procedures for subsequential transducers.

```
136 pseudoMinimal_SSFSTI : SSFSTI → SSFSTI;
137 pseudoMinimal_SSFSTI(Σ, Q, q_0, F, δ, λ, ι, Ψ) :=
    (Σ, R(Q), R(q_0), R(F), δ', λ', ι, Ψ'), where
138     Σ' := {(σ, λ(q, σ)) | ((q, σ), r) ∈ δ};
139     Δ : STATE × (SYMBOL × WORD) → STATE;
140     Δ := {((q, (σ, λ(q, σ))), r) | ((q, σ), r) ∈ δ};
141     R ∈ EQREL;
142     k ∈ ℕ;
143     (R, k) := induction
144         step 0 :
145             X := Proj_2(Ψ) as WORD*;
146             f : STATE → ℕ;
147             f(q) := { #_X(Ψ(q))   if q ∈ F
                        { 0           otherwise;
148             k^(0) := 0;
149             R^(0) := ker(Q, f);
150         step n + 1 :
151             f : (SYMBOL × WORD) × STATE → ℕ;
152             f(c, q) := { R^(n)(Δ(q, c))   if ! Δ(q, c)
                           { 0               otherwise;
153             k^(n+1) := |Proj_2(R^(n))| ;
154             R^(n+1) :=
    intersect_EQREL(Q, R^(n), (intersect_EQREL(Q))(⟨ker(Q, f(c)) | c ∈ Σ'⟩));
155         until (Σ' = ∅) ∨ (k^(n) = |Proj_2(R^(n))|)
156         ;
157     δ' := {((R(q), σ), R(r)) | ((q, (σ, l)), r) ∈ Δ};
158     λ' := {((R(q), σ), l) | ((q, (σ, l)), r) ∈ Δ};
159     Ψ' := {(R(q), P) | (q, P) ∈ Ψ};
160     ;
161 minimal_SSFSTI : SSFSTI → SSFSTI;
162 minimal_SSFSTI(T) := pseudoMinimal_SSFSTI(canonical_SSFSTI(T));
```

Recall that for pseudo-minimization pairs $(\sigma, \lambda(q, \sigma))$ are considered as "artificial letters" of a new alphabet Σ' (Line 138). These "letters" have type $\mathcal{SYMBOL} \times \mathcal{WORD}$. Following Proposition 5.5.18 and Corollary 5.5.19 the pseudo-minimization algorithm proceeds with the minimization exactly as in Program 8.1.14, with the following differences: first, "artifical letters" are used

to define equivalence relations (Lines 151, 152). Second, the initial partitioning is based on the state outputs Ψ (Lines 145-147). The minimization algorithm first converts the transducer in canonical form and then applies pseudo-minimization afterwards (Lines 161-162).

C(M) program for traversing a subsequential transducer

Program 8.3.8 The following function C_λ realizes the generalized transition output function λ^* of a subsequential transducer, given the transition function δ and the transition output function λ.

```
163  C_λ : (TRANSITIONOUT × DTRANSITIONS)×
                STATE × WORD → WORD;
164  C_λ((λ, δ), p, α) := β, where
165      q ∈ STATE;
166      (β, q) := induction
167          step 0 :
168              (β^(0), q^(0)) := (ε, p);
169          step i + 1 :
170              β^(i+1) := β^(i) · λ(q^(i), (α)_{i+1});
171              q^(i+1) := δ(q^(i), (α)_{i+1});
172          until i = |α|
173          ;
174      ;
```

The algorithm uses a simple inductive construction similarly as in Program 8.1.15. Starting from the given first state p and the empty word as a base (Line 168) we concatenate to the current output the output through λ for the current state with the next symbol from α (Line 170). Then we proceed to the next state by making the transition with the following symbol of the input word α (Line 171). The induction ends when all symbols from α are consumed (Line 172). Note that C_λ is not defined if for some state on the path the transition function with the following symbol of α is not defined. In this case the execution of the program will cause an error message.

The above program can be used for obtaining the output of the transducer for a given input word. If $T = (\Sigma, Q, q_0, F, \delta, \lambda, \Psi)$ and α is a word in $L_{\times 2}(T)$, then $O_T(\alpha) = C_\lambda((\lambda, \delta), q_0, \alpha) \cdot \Psi(C_\delta(\delta, q_0, \alpha))$. See Program 8.1.15 for the function C_δ.

C(M) program for composing subsequential transducers

Program 8.3.9 The following algorithm constructs a subsequential transducer that represents the **composition** of two subsequential transducers following Proposition 5.1.10.

```
175  compose_SSFST : SSFST × SSFST → SSFST;
176  compose_SSFST((Σ1, Q1, q01, F1, δ1, λ1, Ψ1), (Σ2, Q2, q02, F2, δ2, λ2, Ψ2)) :=
             T, where
```

```
177   P ∈ (STATE × STATE)*;
178   (T, P) := induction
179     step 0 :
180       T^(0) := (Σ_1, ∅, 1, ∅, ∅, ∅, ∅);
181       P^(0) := ⟨(q_01, q_02)⟩;
182     step n + 1 :
183       (q_1, q_2) := (P^(n))_{n+1};
184       N := {((σ, o), (p_1, p_2)) | σ ∈ Σ_1 & !δ_1(q_1, σ), p_1 = δ_1(q_1, σ),
                l = λ_1(q_1, σ) & |stateseq(δ_2, q_2, l)| = |l| + 1,
                p_2 = C_δ(δ_2, q_2, l), o = C_λ((λ_2, δ_2), q_2, l)};
185       P^(n+1) := P^(n) · ⟨p | p ∈ Proj_2(N) & p ∉ P^(n)⟩;
186       T^(n+1) := (Σ, Q, q_0, F', δ', λ', Ψ'), where
187         (Σ, Q, q_0, F, δ, λ, Ψ) := T^(n);
188         δ' := δ ∪ {((n + 1, σ), #_{P^(n+1)}(q)) | ((σ, o), q) ∈ N};
189         λ' := λ ∪ {((n + 1, σ), o) | ((σ, o), q) ∈ N};
190         (F', Ψ') :=
191           case (q_1 ∈ F_1)
                ∧(|stateseq(δ_2, q_2, Ψ_1(q_1))| = |Ψ_1(q_1)| + 1)
                ∧ (C_δ(δ_2, q_2, Ψ_1(q_1)) ∈ F_2) :
                (F ∪ {n + 1}, Ψ ∪ {(n + 1, o')}), where
192             l := Ψ_1(q_1);
193             q'_2 := C_δ(δ_2, q_2, l);
194             o' := C_λ((λ_2, δ_2), q_2, l) · Ψ_2(q'_2);
195           otherwise : (F, Ψ)
196           ;
197       ;
198     until n = |P^(n)|
199     ;
200   ;
```

The construction is similar to Program 8.2.6. We apply the product construction. The differences are the definition of N in Line 184 and the definition of F and Ψ in Lines 190-196. As in earlier product constructions, the set N yields a description of all transitions departing from the source state pair (q_1, q_2). Here each transition is described as a pair with label (σ, o) and target state pair (p_1, p_2). Label and target state pair are defined following the definitions of δ and λ in Proposition 5.1.10.

Phonetization of numbers as an application

In order to demonstrate the use of the above constructions we present a program for constructing a minimal subsequential transducer that maps a number written as a sequence of digits to its phonetization. This kind of functionality is needed, e.g., for speech synthesis.

Example 8.3.10 For the current implementation we use the phoneme set of the Carnegie Mellon University pronouncing dictionary[2]. In our example we

[2] http://www.speech.cs.cmu.edu/cgi-bin/cmudict

limit the input range to numbers between 1 and 999999. As a matter of fact this segment could be easily extended.

```
1  import ← "Section_8.3.cm";
2  zero := FST_2WORD("0","");
3  zerozero := FST_2WORD("00","");
4  from1to9 := union_FST(⟨FST_2WORD("1","W AH1 N "),
       FST_2WORD("2","T UW1 "),
       FST_2WORD("3","TH R IY1 "),
       FST_2WORD("4","F AO1 R "),
       FST_2WORD("5","F AY1 V "),
       FST_2WORD("6","S IH1 K S "),
       FST_2WORD("7","S EH1 V AH0 N "),
       FST_2WORD("8","EY1 T "),
       FST_2WORD("9","N AY1 N ")⟩);
5  teens := union_FST(⟨FST_2WORD("10","T EH1 N "),
       FST_2WORD("11","IH0 L EH1 V AH0 N "),
       FST_2WORD("12","T W EH1 L V "),
       FST_2WORD("13","TH ER1 T IY1 N "),
       FST_2WORD("14","F AO1 R T IY1 N "),
       FST_2WORD("15","F IH0 F T IY1 N "),
       FST_2WORD("16","S IH0 K S T IY1 N "),
       FST_2WORD("17","S EH1 V AH0 N T IY1 N "),
       FST_2WORD("18","EY0 T IY1 N "),
       FST_2WORD("19","N AY1 N T IY1 N ")⟩);
6  tens := union_FST(⟨FST_2WORD("2","T W EH1 N T IY0 "),
       FST_2WORD("3","TH ER1 D IY0 "),
       FST_2WORD("4","F AO1 R T IY0 "),
       FST_2WORD("5","F IH1 F T IY0 "),
       FST_2WORD("6","S IH1 K S T IY0 "),
       FST_2WORD("7","S EH1 V AH0 N T IY0 "),
       FST_2WORD("8","EY1 T IY0 "),
       FST_2WORD("9","N AY1 N T IY0 ")⟩);
7  hundred := FST_2WORD("","HH AH1 N D R AH0 D ");
8  thousand := FST_2WORD("","TH AW1 Z AH0 N D ");
9  from10to99 := union_FST(teens, concat_FST(tens, from1to9));
10 from1to99 := union_FST(⟨from1to9, teens, concat_FST(tens, from1to9)⟩);
11 from00to99 :=
   union_FST(⟨zerozero, concat_FST(zero, from1to9), from10to99⟩);
12 from100to999 := concat_FST(⟨from1to9, hundred, from00to99⟩);
13 from1to999 := union_FST(from1to99, from100to999);
14 from000to999 := union_FST(concat_FST(zero, from00to99), from100to999);
15 from1000to999999 := concat_FST(⟨from1to999, thousand, from000to999⟩);
16 from1to999999 := union_FST(from1to999, from1000to999999);
17 (N_SSFST, BV) := ssfst_FST(from1to999999);
18 Numbers := minimal_SSFSTI(SSFSTI_SSFST(N_SSFST));
```

After importing the programs from the current section in Line 1 we define in Line 2 and 3 the mapping of one and two zeros to the empty word. Afterwards, in Lines 4,5 and 6 we define the phonetization of single digit numbers, the

numbers between 10 and 19, and the mappings of the tens $(20, 30, \ldots, 90)$ to their corresponding pronunciations. In Lines 7,8 the mapping of the empty word to the pronunciation of hundred and thousand is defined. Then in the following Lines 9-16 we define the mappings of the numbers from 10 to 99, from 1 to 99, from 00 to 99, from 100 to 999, from 1 to 999, from 000 to 999, from 1.000 to 999.999, and finally from 1 to 999.999. In Line 17 we construct the subsequential transducer for the numbers from 1 to 999.999. At the end we obtain the minimal subsequential transducer in Line 18. The resulting subsequential transducer has approximately 52.000 states and 480.000 transitions.

Remark 8.3.11 As noted before, the presented programs are not optimized and therefore the above construction can run out of memory when executed on a personal computer. The minimal subsequential transducer for a finite function can be constructed much more efficiently applying the algorithms presented in Chapter 10.

Remark 8.3.12 Subsequential transducers for defining the mappings between phoneme sequences, English words and the corresponding language probabilities are widely used for implementing probabilistic language models, e.g., in speech recognition applications [Mohri et al., 2008].

8.4 *C(M)* programs for bimachines

In this section we present the algorithms for construction, composition and normalization of bimachines. We assume that all functions defined in Sections 8.1, 8.2 and 8.3 are available.

Translating functional transducers into bimachines

In Section 6.2 we have seen how to translate functional transducers into bimachines. We now give an an implementation.

Program 8.4.1 The following algorithm converts a **functional transducer into a bimachine** following the construction given in the proof of Proposition 6.2.5.

```
1  import ← "Section_8.3.cm";
2  BMOUTPUT is STATE × SYMBOL × STATE → WORD;
3  BM is DFSA × DFSA × BMOUTPUT;
4  bm_FST : FST → BM;
5  bm_FST(A) := ((Σ, {1,...,|P_L|}, 1, {1,...,|P_L|}, δ_L), (Σ, {1,...,|P_R|},
                 1, {1,...,|P_R|}, δ_R), ψ), where
6     ((Σ, Q, I, F, Δ), W, inf) := realTime_FST(A);
7     Δ'' := F_{4→(2,1)}({(q, (u)_1, v, p) | (q, (u, v), p) ∈ Δ});
8     (P_R, δ_R) ∈ (2^STATE)* × DTRANSITIONS;
9     (P_R, δ_R) := induction
10       step 0 :
11          P_R^(0) := ⟨F⟩;
12          δ_R^(0) := ∅;
```

```
13          step n + 1 :
14              N := ⋃(Δ''((P_R^(n))_{n+1}));
15              N' := F_{1→2}(N);
16              P_R^(n+1) := P_R^(n) · ⟨q | q ∈ Proj_2(N') & q ∉ P_R^(n)⟩;
17              δ_R^(n+1) := δ_R^(n) ∪ {((n + 1, c), #_{P_R^(n+1)}(q)) | (c, q) ∈ N'};
18          until n = |P_R^(n)|
19          ;
20      δ'_R := F_{3→(1,2)}(δ_R as 2^{STATE×SYMBOL×STATE});
21      LSYSTEM as 2^{STATE} × (STATE → STATE);
22      Δ' := F_{(2,1)→(4,3)}({(q, (u)_1, v, q') | (q, (u, v), q') ∈ Δ});
23      (P_L, δ_L) ∈ LSYSTEM* × DTRANSITIONS;
24      (P_L, δ_L, ψ) := induction
25          step 0 :
26              δ_L^(0) := ∅;
27              ψ^(0) := ∅;
28              φ_0 : STATE → STATE;
29              φ_0 := {(R, elementOf({q | q ∈ I & q ∈ (P_R)_R})) | R ∈
                                {1, ..., |P_R|} & (P_R)_R ∩ I ≠ ∅};
30              P_L^(0) := ⟨(I, φ_0)⟩;
31          step k + 1 :
32              (L, φ) := (P_L^(k))_{k+1};
33              N := {(a, (L', φ'(a))) | a ∈ Σ, L' = Proj_1(⋃((Δ'(a))(L)))},
                        where
34                  φ' : SYMBOL → (STATE → STATE);
35                  φ'(a) := {(R', q') | (R, q) ∈ φ & ! δ'_R(R), (R', a) ∈ δ'_R(R),
                (q', v) = elementOf({(q', v) | (q', v) ∈ Δ'(a, q) & q' ∈ (P_R)_{R'}})};
36                  ;
37              ψ^(k+1) := ψ^(k) ∪ {((k + 1, a, R'), v) | (a, (L', φ')) ∈ N,
                            (R', q') ∈ φ', q = φ(δ_R(R', a)), (q', v) ∈ Δ'(a, q)};
38              P_L^(k+1) :=
                    P_L^(k) · ⟨(L', φ') | (L', φ') ∈ Proj_2(N) & (L', φ') ∉ P_L^(k)⟩;
39              δ_L^(k+1) :=
                    δ_L^(k) ∪ {((k + 1, a), #_{P_L^(k+1)}((L', φ'))) | (a, (L', φ')) ∈ N};
40          until k = |P_L^(k)|
41          ;
42      ;
```

After importing in Line 1 the programs from the previous section, in Line 2 the type of the bimachine output function is defined. It maps a state of the left automaton, a symbol and a state of the right automaton to an output word. In Line 3 the type of the bimachine is defined as a triple consisting of the left deterministic finite-state automaton, the right deterministic finite-state automaton, and the bimachine output function. Starting from Line 4 the function $\mathrm{bm_{FST}}$ is defined. It takes as input a finite-state transducer. Then it produces the corresponding real-time transducer in Line 6. In Line 7 we define the function Δ'' to map a destination state p of the transducer to the set of pairs consisting of transition input label and source state for transitions to p in the input

transducer. Using Δ'' we perform a determinization of the reversed underlying automaton (Lines 8-19) in a similar way as in Program 8.1.9. According to Proposition 6.2.5 this automaton corresponds to the right automaton of the bimachine. In Line 20 the reverse of the transition function of the right bimachine automaton is constructed in order to map a destination state (a set of transducer states R) to the set of pairs consisting of the source state (a set R' of a-successors of states in R in the transducer) and label a in the right automaton. In Line 21 we define the type of the states for the left automaton of the bimachine. Recall that each state of the left deterministic automaton is given by a set of states and a state selector function (cf. proof of Proposition 6.2.5). Then in Line 22 we define the function Δ' that maps an input label and a source state of the transducer to the set of pairs consisting of destination state and transition output in the transducer. Using Δ' we apply a special determinization to the underlying automaton of the transducer in Lines 23-41, which yields the left deterministic automaton of the bimachine and the bimachine output function. The determinization proceeds similarly as in Program 8.1.9 with a specialised procedure for deriving the successor states and the addition of the construction of the bimachine output function. The initial state $P_L^{(0)}$ of the left deterministic automaton has the form (I, ϕ_0) (Line 30). I is the set of initial states of the transducer and the state selector function ϕ_0 selects any member of $I \cap R$ for a state R of the right deterministic automaton whenever this set is non-empty. (Line 29, the complex notation $(P_R)_R$ is due to the fact that states of the right deterministic automaton are formally represented as numbers). At this point, the output function is empty (Line 27). At the induction step we consider a state (L, ϕ) of the left deterministic automaton. The set N describes all transitions departing from (L, ϕ) in terms of the label a and the target state $(L', \phi'(a))$. The definition of the a-successor state $(L', \phi'(a))$ in Lines 33-36 and the construction of the bimachine output function for (L, ϕ) in Line 37 follow the construction in the proof of Proposition 6.2.5. Finally, the resulting automaton and output function correspond to the left automaton and the output function of the bimachine.

Pseudo-minimization of bimachines

Program 8.4.2 The next algorithm constructs a **pseudo-minimal bimachine** in accordance with Definition 6.3.3 by minimizing the left and the right automaton of the bimachine (considered as coloured automata using state profiles as colours) following the procedure described in Section 6.3.

43 $\text{pseudoMinimal}_{\text{BM}} : \mathcal{BM} \to \mathcal{BM}$;
44 $\text{pseudoMinimal}_{\text{BM}}((\Sigma_L, L, s_L, F_L, \delta_L), (\Sigma_R, R, s_R, F_R, \delta_R), \psi) :=$
$((\Sigma_L, \text{R}_\text{L}(L), \text{R}_\text{L}(s_L), \emptyset, \delta'_L), (\Sigma_R, \text{R}_\text{R}(R), \text{R}_\text{R}(s_R), \emptyset, \delta'_R), \psi')$, **where**
45 $R_L \in \mathcal{EQRECL}$;
46 $k_L \in \mathbb{N}$;
47 $(R_L, k_L) :=$ **induction**
48 **step** 0 :
49 $\Psi_\text{L} : \mathcal{STATE} \to 2^{\mathcal{SYMBOL} \times \mathcal{STATE} \times \mathcal{WORD}}$;
50 $\Psi_L := \mathcal{F}_{1\to(2,3,4)}(\psi$ **as** $2^{\mathcal{STATE} \times \mathcal{SYMBOL} \times \mathcal{STATE} \times \mathcal{WORD}})$;
51 $\text{f} : \mathcal{STATE} \to \mathbb{N}$;

52 $X := \mathrm{Proj}_2(\Psi_L)$ **as** $(2^{\mathcal{SYMBOL}\times\mathcal{STATE}\times\mathcal{WORD}})^*$;
53 $\mathrm{f}(q) := \begin{cases} \#_X(\Psi_{\mathrm{L}}(q)) & \textbf{if } !\,\Psi_{\mathrm{L}}(q) \\ 0 & \textbf{otherwise;} \end{cases}$
54 ${k_L}^{(0)} := 0$;
55 ${R_L}^{(0)} := \ker(L, f)$;
56 **step** $n+1$:
57 $\mathrm{f} : \mathcal{SYMBOL} \times \mathcal{STATE} \to \mathbb{N}$;
58 $\mathrm{f}(c,q) := \begin{cases} {\mathrm{R_L}}^{(\mathrm{n})}(\delta_{\mathrm{L}}(q,c)) & \textbf{if } !\,\delta_{\mathrm{L}}(q,c) \\ 0 & \textbf{otherwise;} \end{cases}$
59 ${k_L}^{(n+1)} := \left|\mathrm{Proj}_2({R_L}^{(n)})\right|$;
60 ${R_L}^{(n+1)} :=$
$\mathrm{intersect}_{\mathrm{EQREL}}(L, (\mathrm{intersect}_{\mathrm{EQREL}}(L))(\langle \ker(L, \mathrm{f}(c)) \mid c \in \Sigma_L \rangle), {R_L}^{(n)})$;
61 **until** ${k_L}^{(n)} = \left|\mathrm{Proj}_2({R_L}^{(n)})\right|$
62 ;
63 $R_R \in \mathcal{EQREL}$;
64 $k_R \in \mathbb{N}$;
65 $(R_R, k_R) :=$ **induction**
66 **step** 0 :
67 $\Psi_{\mathrm{R}} : \mathcal{STATE} \to 2^{\mathcal{SYMBOL}\times\mathcal{STATE}\times\mathcal{WORD}}$;
68 $\Psi_R := \mathcal{F}_{3\to(2,1,4)}(\psi$ **as** $2^{\mathcal{STATE}\times\mathcal{SYMBOL}\times\mathcal{STATE}\times\mathcal{WORD}})$;
69 $\mathrm{f} : \mathcal{STATE} \to \mathbb{N}$;
70 $X := \mathrm{Proj}_2(\Psi_R)$ **as** $(2^{\mathcal{SYMBOL}\times\mathcal{STATE}\times\mathcal{WORD}})^*$;
71 $\mathrm{f}(q) := \begin{cases} \#_X(\Psi_{\mathrm{R}}(q)) & \textbf{if } !\,\Psi_{\mathrm{R}}(q) \\ 0 & \textbf{otherwise;} \end{cases}$
72 ${k_R}^{(0)} := 0$;
73 ${R_R}^{(0)} := \ker(R, f)$;
74 **step** $n+1$:
75 $\mathrm{f} : \mathcal{SYMBOL} \times \mathcal{STATE} \to \mathbb{N}$;
76 $\mathrm{f}(c,q) := \begin{cases} {\mathrm{R_R}}^{(\mathrm{n})}(\delta_{\mathrm{R}}(q,c)) & \textbf{if } !\,\delta_{\mathrm{R}}(q,c) \\ 0 & \textbf{otherwise;} \end{cases}$
77 ${k_R}^{(n+1)} := \left|\mathrm{Proj}_2({R_R}^{(n)})\right|$;
78 ${R_R}^{(n+1)} :=$
$\mathrm{intersect}_{\mathrm{EQREL}}(R, (\mathrm{intersect}_{\mathrm{EQREL}}(R))(\langle \ker(R, \mathrm{f}(c)) \mid c \in \Sigma_R \rangle), {R_R}^{(n)})$;
79 **until** ${k_R}^{(n)} = \left|\mathrm{Proj}_2({R_R}^{(n)})\right|$
80 ;
81 $\delta'_L := \{((\mathrm{R_L}(q), \sigma), \mathrm{R_L}(r)) \mid ((q,\sigma), r) \in \delta_L\}$;
82 $\delta'_R := \{((\mathrm{R_R}(q), \sigma), \mathrm{R_R}(r)) \mid ((q,\sigma), r) \in \delta_R\}$;
83 $\psi' := \{((\mathrm{R_L}(l), a, \mathrm{R_R}(r)), s) \mid ((l,a,r), s) \in \psi\}$;
84 ;

The algorithm proceeds by first finding the Myhill-Nerode relation R_L for the left coloured automaton according to Corollary 6.3.2 (Lines 45-62). In Lines 49-55 the induction is initialized with $R_L^{(0)} := ker_L(\psi_L)$. Afterwards (Lines 56-62) we proceed exactly as in Program 8.1.14. The Myhill-Nerode relation R_R for the right coloured automaton is constructed in the corresponding dual way in Lines 63-80. Afterwards we define the transition functions of the minimized left

and right automata (Lines 81, 82). At the end we define the bimachine output function in Line 83.

Direct composition of bimachines

As a last point in this chapter we consider the composition of two bimachines. We assume that the alphabets of the two machines are identical.

Program 8.4.3 The algorithm computes the **composition of two bimachines** following the construction presented in Section 6.4.

85 $\mathrm{C}_\Psi : (\mathcal{BMOUTPUT} \times \mathcal{DTRANSITIONS} \times \mathcal{DTRANSITIONS}) \times \mathcal{STATE} \times \mathcal{WORD} \times \mathcal{STATE} \to \mathcal{WORD}$;

86 $\mathrm{C}_\Psi((\Psi, \delta_L, \delta_R), l, \alpha, r) := \beta$, **where**

87 $\quad \pi_L := \mathrm{stateseq}(\delta_L, l, \alpha)$;

88 $\quad \pi_R := \mathrm{stateseq}(\delta_R, r, \rho(\alpha))$;

89 $\quad \beta := \bigodot(\langle \Psi((\pi_L)_i, (\alpha)_i, (\pi_R)_{|\alpha|+1-i}) \mid i \in \{1, \ldots, |\alpha|\}\rangle)$;

90 $\quad$;

91 $\mathrm{compose}_{\mathrm{BM}} : \mathcal{BM} \times \mathcal{BM} \to \mathcal{BM}$;

92 $\mathrm{compose}_{\mathrm{BM}}(((\Sigma'_L, L', s'_L, F'_L, \delta'_L), (\Sigma'_R, R', s'_R, F'_R, \delta'_R), \psi'),$
$\quad ((\Sigma''_L, L'', s''_L, F''_L, \delta''_L), (\Sigma''_R, R'', s''_R, F''_R, \delta''_R), \psi'')) :=$
$((\Sigma, \{1, \ldots, |P_L|\}, 1, \{1\}, \delta_L), (\Sigma, \{1, \ldots, |P_R|\}, 1, \{1\}, \delta_R), \psi)$, **where**

93 $\quad \Sigma := \Sigma'_L \cap \Sigma'_R \cap \Sigma''_L \cap \Sigma''_R$;

94 $\quad \delta'_{R_0} := \mathcal{F}_{(2,3)\to 1}(\delta'_R$ **as** $2^{\mathcal{STATE}\times\mathcal{SYMBOL}\times\mathcal{STATE}})$;

95 $\quad \delta'_{L_0} := \mathcal{F}_{(2,3)\to 1}(\delta'_L$ **as** $2^{\mathcal{STATE}\times\mathcal{SYMBOL}\times\mathcal{STATE}})$;

96 $\quad \mathrm{succL}, \mathrm{succR} : \mathcal{SYMBOL} \times (\mathcal{STATE} \times \mathcal{STATE} \times \mathcal{STATE}) \to 2^{\mathcal{STATE}\times\mathcal{STATE}\times\mathcal{STATE}}$;

97 $\quad \mathrm{succL}(a, (l'_1, r'_1, l''_1)) :=$
$$\begin{cases} \{(\delta'_{\mathrm{L}}(l'_1, a), r'_2, (\mathrm{C}_\delta(\delta''_L))(l''_1, \psi'(l'_1, a, r'_2))) \mid r'_2 \in \delta'_{\mathrm{R}_0}(a, r'_1)\} & \textbf{if } !\,\delta'_{\mathrm{R}_0}(a, r'_1) \\ \emptyset & \textbf{otherwise;} \end{cases}$$

98 $\quad \mathrm{succR}(a, (l'_2, r'_2, r''_2)) :=$
$$\begin{cases} \{(l'_1, \delta'_{\mathrm{R}}(r'_2, a), (\mathrm{C}_\delta(\delta''_R))(r''_2, \rho(\psi'(l'_1, a, r'_2)))) \mid l'_1 \in \delta'_{\mathrm{L}_0}(a, l'_2)\} & \textbf{if } !\,\delta'_{\mathrm{L}_0}(a, l'_2) \\ \emptyset & \textbf{otherwise;} \end{cases}$$

99 $\quad P_L \in (2^{\mathcal{STATE}\times\mathcal{STATE}\times\mathcal{STATE}})^*$;

100 $\quad \delta_L \in \mathcal{DTRANSITIONS}$;

101 $\quad (P_L, \delta_L) :=$ **induction**

102 $\quad\quad$ **step** 0 :

103 $\quad\quad\quad {P_L}^{(0)} := \langle \{s'_L\} \times R' \times \{s''_L\}\rangle$;

104 $\quad\quad\quad {\delta_L}^{(0)} := \emptyset$;

105 $\quad\quad$ **step** $k+1$:

106 $\quad\quad\quad N := \{(a, \bigcup((\mathrm{succL}(a))((P_L{}^{(k)})_{k+1}))) \mid a \in \Sigma\}$;

107 $\quad\quad\quad {P_L}^{(k+1)} := {P_L}^{(k)} \cdot \langle q \mid q \in \mathrm{Proj}_2(N)\ \&\ q \notin {P_L}^{(k)}\rangle$;

108 $\quad\quad\quad {\delta_L}^{(k+1)} := {\delta_L}^{(k)} \cup \{((k+1, a), \#_{{P_L}^{(k+1)}}(q)) \mid (a, q) \in N\}$;

109 $\quad\quad$ **until** $k = \left|{P_L}^{(k)}\right|$

110 $\quad\quad$;

111 $\quad P_R \in (2^{\mathcal{STATE}\times\mathcal{STATE}\times\mathcal{STATE}})^*$;

112 $\quad \delta_R \in \mathcal{DTRANSITIONS}$;

113 $\quad (P_R, \delta_R) :=$ **induction**

114 $\quad\quad$ **step** 0 :

```
115             P_R^(0) := ⟨L' × {s'_R} × {s''_R}⟩;
116             δ_R^(0) := ∅;
117         step k + 1 :
118             N := {(a, ⋃((succR(a))((P_R^(k))_{k+1}))) | a ∈ Σ};
119             P_R^(k+1) := P_R^(k) · ⟨q | q ∈ Proj_2(N) & q ∉ P_R^(k)⟩;
120             δ_R^(k+1) := δ_R^(k) ∪ {((k + 1, a), #_{P_R^(k+1)}(q)) | (a, q) ∈ N};
121         until k = |P_R^(k)|
122         ;
123     ψ ∈ BMOUTPUT;
124     ψ := {((l, a, r), ψ_0(l, a, r)) | l ∈ {1, ..., |P_L|}, a ∈ Σ, r ∈
    {1, ..., |P_R|}}, where
125         ψ_0 ∈ BMOUTPUT;
126         ψ_0(l, a, r) := (C_Ψ(ψ'', δ''_L, δ''_R))(l''_1, ψ'(l'_1, a, r'_2), r''_2), where
127             l'_1 := Proj_1(elementOf((P_L)_l));
128             r'_2 := Proj_2(elementOf((P_R)_r));
129             l'_2 := δ'_L(l'_1, a);
130             r'_1 := δ'_R(r'_2, a);
131             l''_1 := elementOf({l''_1 | (l'_1, r'_1, l''_1) ∈ (P_L)_l});
132             r''_2 := elementOf({r''_2 | (l'_2, r'_2, r''_2) ∈ (P_R)_r});
133             ;
134         ;
135     ;
```

The program starts with the definition of the generalized bimachine output function C_ψ in Lines 85-90. The symbol $\bigodot$ denotes the concatenation of a list of words. For computing the left and right deterministic automata of the target bimachine two inductive power set constructions are used. Starting from the initial states of the deterministic automata (Lines 103, 115) we compute the list of states needed for the two automata. In Lines 94-98 auxiliary functions are defined based on the first input bimachine $\mathcal{B}_1$. Here δ'_{R_0} computes for a symbol a and state q of the right automaton of $\mathcal{B}_1$ the set of all a-predecessors of q. Similarly δ'_{L_0} computes for a symbol a and state q of the left automaton of $\mathcal{B}_1$ the set of all a-predecessors of q. With the help of these functions then in Lines 97 and 98 the sets of a-successors of single triples (l'_1, r'_1, l''_2) respectively (l'_2, r'_2, r''_2) with respect to the relations Δ_L and Δ_R introduced in the formal construction in Section 6.4 are computed. After this preparation the inductive power set constructions are defined. In Lines 106 and 118 the sets N provide an implicit description of all trannsitions from the states $(P_L{}^{(k)})_{k+1}$ and $(P_R{}^{(k)})_{k+1}$ that are treated at step $k+1$ of the two inductions. At this point we use the a-successor sets for single triples. Given the sets N, we find the new states needed (Lines 107, 119) and the new transitions needed (Lines 108, 120). Finally in Lines 123-134 we define the bimachine output function according to the formal construction.

Note that the generalized bimachine output function C_ψ can be used for implementing the output function of the bimachine. If

$$B = \langle \langle \Sigma, Q_L, s_L, F_L, \delta_L \rangle, \langle \Sigma, Q_R, s_R, F_R, \delta_R \rangle, \Psi \rangle,$$

then $O_B(\alpha) = C_\Psi((\Psi, \delta_L, \delta_R), s_L, \alpha, s_R)$.

Bignum arithmetics with bimachines as an application

We complete the section with an example presenting the use of bimachines for implementing basic arithmetic operations on unbounded natural numbers.

Example 8.4.4 Any natural number (e.g. 5389) can be represented as string of digits such as `['5','3','8','9']`. The programs defined below take as input an arbitrary natural number $K \in \mathbb{N}$. Given the number K an "addition" bimachine is computed that reads as input a second natural number x represented as a string of digits as indicated above. The output of the addition bimachine is the representation of $x + K$ as a string of digits. In a similar way, for a given input number K, bimachines for the operations $x \mapsto x - K$ (subtraction, defined for inputs $x \geq K$), $x \mapsto x \cdot K$ (multiplication), $x \mapsto x/K$ (division), and for the remainder operation after dividing by K are computed. In each case, input and output numbers are represented as strings.

The program starts with importing all programs from the current section in Line 1. The bimachines representing the aforementioned operations are obtained from corresponding transducers that are built first. In general these transducers are non-deterministic, conversion to a bimachine leads to a deterministic device. The types of the new functions for computing the operations are defined in Lines 2-3. The functions take as input a natural number and compute a transducer or a bimachine. Afterwards we proceed with the definition of the set of digits in Line 4 and the definition of the function preventLeadingZeros, which restricts the domain of a given transducer to the set of all numbers expressed without leading zeros (Lines 5-8). In Lines 9-10 we define the function for converting the code (UNICODE) of the symbol for a digit to the corresponding number and in Lines 11-13 we define the inverse function.

```
1  import ← "Section_8.4.cm";
2  addT_K, subT_K, mulT_K, divExactT_K, divT_K, remT_K : ℕ → 𝓕𝓢𝓣;
3  addB_K, subB_K, mulB_K, divB_K, remB_K : ℕ → ℬℳ;
4  Digits := {'0', ..., '9'};
5  preventLeadingZeros : 𝓕𝓢𝓣 → 𝓕𝓢𝓣;
6  preventLeadingZeros(T) := compose_FST(identity_FSA(Dom), T), where
7      Dom := union_FSA(FSA_WORD("0"),
           concat_FSA(symbolSet2FSA({'1', ..., '9'}),
           star_FSA(symbolSet2FSA(Digits))));
8      ;
9  digit2number : 𝓢𝓨ℳℬ𝓞ℒ → ℕ;
10 digit2number(d) := d −' '0';
11 number2digit : ℕ → 𝓢𝓨ℳℬ𝓞ℒ;
12 number2digit(n) := n +' '0';
```

Addition. The first operation to be realized is the addition $x \mapsto x + K$. It is hard to design a mechanical procedure for digit-based addition when reading the sequences of digits from left to right. Hence first an "addition" transducer for the reverse reading order is built. Using the reverse operation for transducers, the transducer for the correct reading order is obtained as a derivate. To illustrate the construction of the transducer for the reverse reading let us assume that $K = 907$. In the transducer to be built, addition of a number x represented as

a sequence of digits, say $x =$ `['5','8','7','7']`, is realized on a path of the following form:

$$907 \rightarrow^{'7':'4'} 91 \rightarrow^{'7':'8'} 9 \rightarrow^{'8':'7'} 1 \rightarrow^{'5':'6'} 0$$

States are numbers, starting with K. Inputs are the digits `'7','7','8','5'` of x in the reverse reading order. When reading a digit d from state K' we add the corresponding number to K'. The last digit of the sum n (i.e. the remainder of the division of n by 10) is the output of the transition. The new number reached is the integer $\lfloor n/10 \rfloor$. Below, these states/numbers are called "remainders". Note that the reversed output is $\mathtt{x} = [\mathtt{'6'},\mathtt{'7'},\mathtt{'8'},\mathtt{'4'}]$, which in fact represents the sum 907+5877. The above path only shows the states/remainders 907, 91, 9, 1, and 0 needed for adding $x = 5877$. However, for any given number K, only a finite number of states are needed for processing any given sequence of digits. This leads to the following program.

```
13  addT_K(K) := reverse_FST(Digits, {1,...,|L|+1}, {1}, {f}, Δ'), where
14      L ∈ ℕ*;
15      Δ ∈ 2^TRANSITION;
16      (Δ, L) := induction
17          step 0 :
18              (L^(0), Δ^(0)) := (⟨K⟩, ∅);
19          step i+1 :
20              c := (L^(i))_{i+1};
21              N := {((⟨d⟩, ⟨d'⟩), c') | d ∈ Digits, n = digit2number(d) + c,
                        c' = n/10, d' = number2digit(n mod 10)};
22              L^(i+1) := L^(i) · ⟨c' | c' ∈ Proj_2(N) & c' ∉ L^(i)⟩;
23              Δ^(i+1) := Δ^(i) ∪ {(i+1, l, #_{L^(i+1)}(c')) | (l, c') ∈ N};
24          until i = |L^(i)|
25          ;
26      f := |L| + 1;
27      Δ' := Δ ∪ {(i, (ε, c), f) | i ∈ {1,...,|L|},
                        c = { ""              if (L)_i = 0
                            { ρ(str((L)_i))   otherwise };
28      ;
```

Lines 14-27 describe the inductive construction of the transducer for the reverse reading order. As illustrated above, each state in the transducer (with the exception of the final state f) corresponds to a given remainder. The "official" name for a state is an index number in the list L. In this list at the i-th position we store the i-th remainder obtained. The list L and the transitions Δ from states i are defined in the induction in Lines 16-27. We start with initializing the list L with the first remainder K in Line 18. The inductive step in Lines 19-23 proceeds with the definition of the transitions for state $i+1$ with remainder c (Line 20). In Line 21 in the set N we collect for each digit d the last digit d' and the remainder c' of the sum with c. The list L is extended adding the new remainders in Line 22 and Δ is extended adding the transitions from $i+1$ in Line 23. The induction ends when the list L is exhausted. Finally in Lines 26-27 we define from each state an outgoing transition to the final state f with ε on the upper tape and the digits for the remainder on the lower tape. Note that

these transitions are only relevant when all input digits are consumed.

Multiplication. The next function $\mathrm{mulT_K}$ constructs for a given number K a transducer representing the function $f(x) = x \times K$.

29 $\mathrm{mulT_K}(K) := \mathrm{reverse_{FST}}(\mathit{Digits}, \{1, \ldots, |L|+1\}, \{1\}, \{f\}, \Delta')$, **where**
30 $L \in \mathbb{N}^*$;
31 $\Delta \in 2^{\mathcal{TTRANSITION}}$;
32 $(\Delta, L) :=$ **induction**
33 **step** 0 :
34 $(L^{(0)}, \Delta^{(0)}) := (\langle 0 \rangle, \emptyset)$;
35 **step** $i+1$:
36 $c := (L^{(i)})_{i+1}$;
37 $N := \{((\langle d \rangle, \langle d' \rangle), c') \mid d \in \mathit{Digits}, n = \mathrm{digit2number}(d) \times K + c,$
$c' = n/10, d' = \mathrm{number2digit}(n \mod 10)\}$;
38 $L^{(i+1)} := L^{(i)} \cdot \langle c' \mid c' \in \mathrm{Proj}_2(N)\ \&\ c' \notin L^{(i)} \rangle$;
39 $\Delta^{(i+1)} := \Delta^{(i)} \cup \{(i+1, l, \#_{L^{(i+1)}}(c')) \mid (l, c') \in N\}$;
40 **until** $i = \left|L^{(i)}\right|$
41 ;
42 $f := |L| + 1$;
43 $\Delta' := \Delta \cup \{(i, (\varepsilon, c), f) \mid i \in \{1, \ldots, |L|\},$
$c = \begin{cases} \text{""} & \textbf{if } (L)_i = 0 \\ \rho(\mathrm{str}((L)_i)) & \textbf{otherwise} \end{cases}\};$
44 ;

The construction used in Lines 29-44 is very similar to the preceding one for $\mathrm{addT_K}$. The only differences are (i) the initial state is initialized to the remainder 0 and (ii) the definition of N in the inductive step (Line 37) is modified, here we multiply with K every digit of the input.

Subtraction. The function $\mathrm{subT_K}$ for subtraction $x \mapsto x - K$ is obtained in a simple way: we just invert the addition transducer $\mathrm{addT_K}(K)$. Before the inversion the domain of the transducer is restricted to numbers without leading zeros. Otherwise the result would not be functional:

45 $\mathrm{subT_K}(K) := \mathrm{inverse_{FST}}(\mathrm{preventLeadingZeros}(\mathrm{addT_K}(K)))$;

Division. Similarly the function $\mathrm{divExactT_K}$ building the transducer representing the function $f(x) = x/K$ for the numbers divisible by K is obtained by inversion of the multiplication transducer:

46 $\mathrm{divExactT_K}(K) := \mathrm{inverse_{FST}}(\mathrm{preventLeadingZeros}(\mathrm{mulT_K}(K)))$;

The next step generalizes the exact division operation to numbers not divisible by K. By composing $\mathrm{subT_K}(i)$ with $\mathrm{divExactT_K}(K)$ we represent the functions $f(x) = (x-i)/K$ for the numbers x for which $x - i$ is divisible by K. In Line 47 we build the union of $\mathrm{divExactT_K}(K)$ with all these functions ($i = 1, \ldots, K-1$) for the representation of the function $f(x) = \lfloor x/K \rfloor$.

47 $\mathrm{divT_K}(K) := \mathrm{union_{FST}}(\langle \mathrm{divExactT_K}(K) \rangle \cdot$
$\langle \mathrm{compose_{FST}}(\mathrm{subT_K}(i), \mathrm{divExactT_K}(K)) \mid i \in \{1, \ldots, K-1\} \rangle)$;

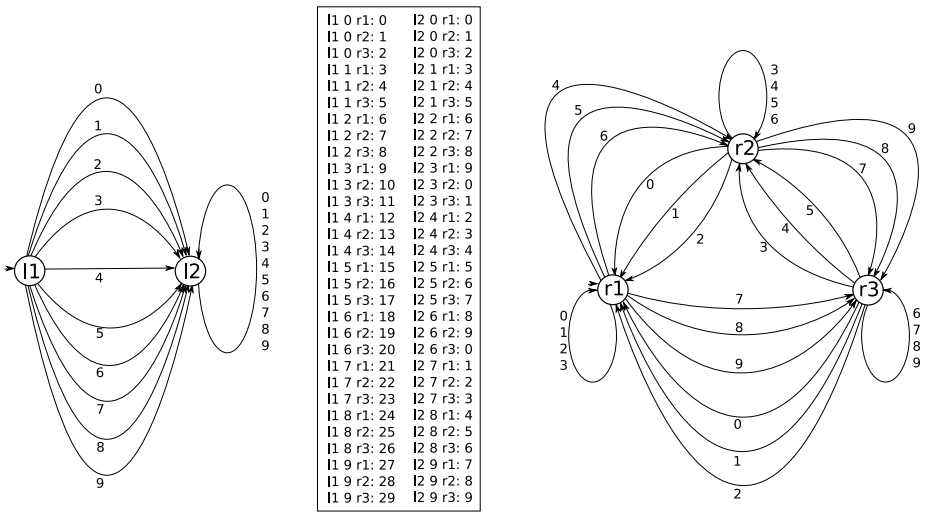

Figure 8.1: The bimachine representing the function $f(x) = x \times 3$.

Remainder of division. The function $\mathrm{remT_K}$ (Lines 48-51) constructs the transducer representing the function $f(x) = x - \lfloor x/K \rfloor \times K$ (remainder of division by K). The range of $\mathrm{multT_K}$ represents all numbers divisible by K (Line 49). The image of this set under the function $\mathrm{addT_K}(i)$ represents the numbers with remainder i of division by K. Each of these sets is mapped to i and the corresponding mappings are joined in Line 48 for constructing the function $\mathrm{remT_K}(K)$.

48 $\mathrm{remT_K}(K) := \mathrm{union_{FST}}(\langle \mathrm{product_{FSA}}(exact, \mathrm{FSA_{WORD}}(\text{"0"}))\rangle \cdot$
$\langle \mathrm{product_{FSA}}(\mathrm{range_{FST}}(\mathrm{compose_{FST}}(ID_{exact}, \mathrm{addT_K}(i))),$
$\mathrm{FSA_{WORD}}(\mathrm{str}(i))) \mid i \in \{1, \ldots, K-1\}\rangle,$ **where**
49 $exact := \mathrm{range_{FST}}(\mathrm{mulT_K}(K));$
50 $ID_{exact} := \mathrm{identity_{FSA}}(exact);$
51 ;

The program ends by defining the corresponding bimachine constructions in Lines 52-56.

52 $\mathrm{addB_K}(K) := \mathrm{pseudoMinimal_{BM}}(\mathrm{bm_{FST}}(\mathrm{addT_K}(K)));$
53 $\mathrm{subB_K}(K) := \mathrm{pseudoMinimal_{BM}}(\mathrm{bm_{FST}}(\mathrm{subT_K}(K)));$
54 $\mathrm{mulB_K}(K) := \mathrm{pseudoMinimal_{BM}}(\mathrm{bm_{FST}}(\mathrm{mulT_K}(K)));$
55 $\mathrm{divB_K}(K) := \mathrm{pseudoMinimal_{BM}}(\mathrm{bm_{FST}}(\mathrm{divT_K}(K)));$
56 $\mathrm{remB_K}(K) := \mathrm{pseudoMinimal_{BM}}(\mathrm{bm_{FST}}(\mathrm{remT_K}(K)));$

In Figure 8.1 the resulting bimachine of $\mathrm{mulB_K}(3)$ is shown.

Remark 8.4.5 For any $K \geq 1$ the functions represented by $\mathrm{divT_K}$ and $\mathrm{remT_K}$ do have the bounded variation property. For some K (e.g. for $K = 2, 5, 10$) the functions represented by $\mathrm{mulT_K}$ do have the bounded variation property. For other K (e.g. for $K = 3, 7, 9$) the functions represented by $\mathrm{mulT_K}$ do not have the bounded variation property. For any $K \geq 1$ the functions represented

by $\mathrm{addT_K}$ and $\mathrm{subT_K}$ do not have the bounded variation property. Therefore many of the above arithmetic functions can not be represented as subsequential transducers.

Remark 8.4.6 Using the bimachine (or transducer) composition we can effectively construct a bimachine for any composition of the presented arithmetic operations (like $f(x) := (\lfloor x/3 \rfloor + 48) \times 256 - 125$). The result is a single bimachine and therefore performing the composed calculation will proceed as fast as a single arithmetic operation. The time will be linear in respect to the number of digits of the input number.

Chapter 9

The Aho-Corasick algorithm

This chapter describes a special construction based on finite-state automata with important applications: the Aho-Corasick Algorithm [Aho and Corasick, 1975] is used to efficiently find all occurrences of a finite set of strings (also called pattern set, or dictionary) in a given input string, called the "text". Search is "online", which means that the input text is neither fixed nor preprocessed in any way. This problem is a special instance of pattern matching in strings, and other automata constructions are used to solve other pattern matching tasks, see e.g. [Mohri, 1995, Navarro and Raffinot, 2002]. From an automaton point of view, the Aho-Corasick Algorithm comes in two variants. We first present the more efficient version where a classical deterministic finite-state automaton is built for text search. The disadvantage of this first construction is that the resulting automaton can become very large, in particular for large pattern alphabets. In Section 9.3 we present the second version, where an automaton with additional transitions of a particular kind is built, yielding a much smaller device for text search.

9.1 Formal construction – first version

Let $D \subseteq \Sigma^*$ denote a finite set of strings, called the *dictionary* or *pattern set*. It is straightforward to describe a *naïve algorithm* for searching all occurrences of strings in the set D in a text $t \in \Sigma^*$: for each possible offset position k in the text t we start reading the suffix s of t that begins at position k. When we find a prefix p of s that represents a string in D we output the hit and continue. As soon as we reach a prefix p of s that does not represent a prefix of any pattern in D, or if we reach the end of t, we continue the search at the new offset position $k' = k + 1$ of t. The naïve algorithm terminates after we have exhausted all possible offset positions in the text.

Due to steps where we return to an earlier text position k', the worst-case time complexity of the naïve algorithm is $O(m \cdot |t|)$, where m is the maximal length of an original string in the D. When using the naïve algorithm, such backward

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

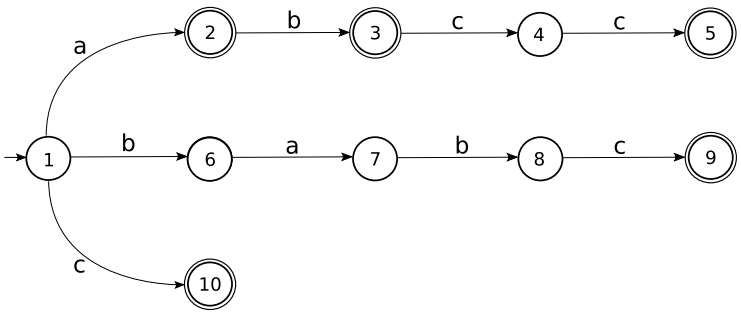

Figure 9.1: The trie for the dictionary $\mathcal{D} = \{a, ab, abcc, babc, c\}$.

steps cannot be avoided.

Below we present a construction introduced in [Aho and Corasick, 1975] for searching all occurrences of strings from D in a given text t in time $O(|t|)$. This algorithm does not use backward steps, and no memory device is needed: in this approach we compute, given the dictionary D, a deterministic finite state automaton $\mathcal{A}$ that recognizes the language $L(\mathcal{A}) = \Sigma^* \cdot D$. Note that input texts are assumed to be strings in Σ^*. The linear time complexity holds under the assumption that we want to output all text positions where an infix of t is ending that represents a string in D.

The construction of the automaton $\mathcal{A}$ represents a preprocessing step that is performed once, given D. If the size of the alphabet Σ is considered as a constant, the number of states and transitions of $\mathcal{A}$ is linear in the size $\|D\|$[1]. With simple optimizations described in the next section also the time and space needed to compute $\mathcal{A}$ is linear in $\|D\|$. Search using $\mathcal{A}$ is more efficient than the naïve algorithm. For illustration purposes we use the dictionary

$$D = \{a, ab, abcc, babc, c\}$$

over the alphabet $\{a, b, c\}$. The construction of the deterministic finite-state automaton for the language $\Sigma^* \cdot D$ proceeds in two steps. The first step is straightforward.

First Step. We construct the trie (cf. Def. 3.1.4) for the dictionary D. Figure 9.1 shows the trie $\mathcal{A}_D$ for the example dictionary D. Obviously the number of non-initial states in the trie $\mathcal{A}_D$ does not exceed the total number of symbols in the list of all dictionary entries. In what follows, following our earlier notation the path label for a trie state q (cf. Definition 3.1.6) is denoted $plab(q)$ and the depth of q is denoted $d(q)$. By δ_D and F_D we respectively denote the transition function and the set of final states of the trie.

Second Step. Given $\mathcal{A}_D$ we construct a deterministic automaton $\mathcal{A}$ with extended set of final states F that represents the language $\Sigma^* \cdot D$. The set of states of $\mathcal{A}_D$ is not modified. The new transition function δ is built in such a way that

1. δ is total - for each state q and each symbol $\sigma \in \Sigma$ always $\delta(q, \sigma)$ is defined.

[1] The size $\|D\|$ of D is the sum of the lengths of the strings in D.

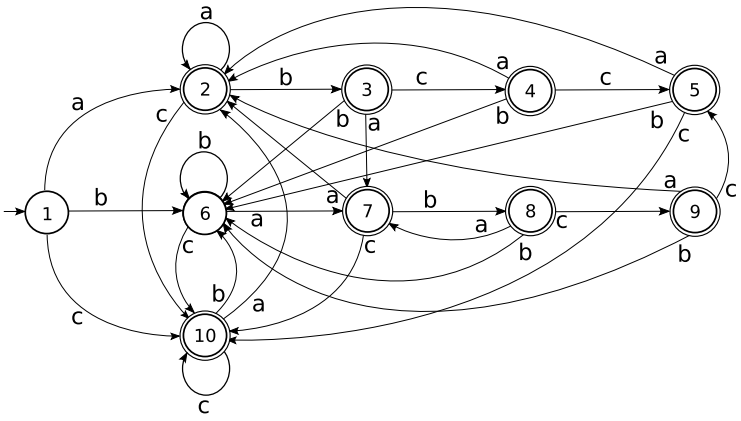

Figure 9.2: The Aho-Corasick automaton for the dictionary $\mathcal{D} = \{a, ab, abcc, babc, c\}$.

2. After reading a (partial) text t the state $q_t := \delta^*(q_0, t)$ "corresponds" to the *longest* prefix u of a dictionary word that represents a suffix of t, in the sense that $plab(q_t) = u$.

Clearly, the first conditions is necessary since we want to be able to process arbitrary texts. The second condition helps to redefine the set of final states in such a way that $q_t := \delta^*(q_0, t)$ is in F iff a suffix of t is an entry of D, see below.

In the algorithm, the states of the trie are considered in a breadth-first manner (i.e., ordered by depth d), extending the trie transition function δ_D and the set of final states F_D. When visiting a state q, we add a new transition from q with each letter $\sigma \in \Sigma$ where $\delta_D(q, \sigma)$ is not defined. All final states in F_D stay final. In some cases (s.b.), a non-final state of $\mathcal{A}_D$ is turned into a final state in F.

For the start state q_0 we just add a new transition leading back to q_0 for each symbol $\sigma \in \Sigma$ where $\delta_D(q_0, \sigma)$ is not defined.

Consider the situation where we visit a state $q \neq q_0$ of depth n. Let $a_1 \cdots a_n := plab(q)$. The *failure state* of q is defined as $p' := \delta^*(q_0, a_2 \cdots a_n)$. Given p' we simply define $\delta(q, \sigma) := \delta(p', \sigma)$ for all $\sigma \in \Sigma$ where $\delta_D(q, \sigma)$ is undefined. If the failure state p' is in the extended set of final states created up to this point, then also q is added to the set F of final states.

Remark 9.1.1 In the situation where we visit a state $q \neq q_0$ with $d(q) = n$ we have already processed all states with depth $< n$ before. This ensures that both $p' = \delta^*(q_0, a_2 \cdots a_n)$ and $\delta(p', \sigma)$ are defined. We shall prove below (cf. Corollary 9.1.3) that the failure state p' of a state q always satisfies $d(p') < d(q)$, which shows that any path from q_0 of length k that includes a new transition leads to a state with depth $< k$.

The resulting deterministic finite-state automaton for our example is shown in Figure 9.2. The correctness of the above algorithm is proved in the next propositions.

Proposition 9.1.2 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be the deterministic finite-state automaton constructed by the above method for the dictionary D. Let $t \in \Sigma^*$ and $q := \delta^*(q_0, t)$. Then $plab(q)$ is the longest suffix of t that is in $Pref(D)$.*

Proof. The proof proceeds by induction on the length of t.
For $|t| = 0$ the proposition is trivial.
As induction hypothesis let us assume that the proposition is correct for all texts t of length k. For the induction step let $\sigma \in \Sigma$ and let $t' := t \cdot \sigma$ be a text of length $k+1$. Consider $q' := \delta^*(q_0, t \cdot \sigma) = \delta(q, \sigma)$. We have two cases.
Case 1: $\delta_D(q, \sigma)$ is defined. Then $q' = \delta(q, \sigma) = \delta_D(q, \sigma)$ and $plab(q') = plab(q) \cdot \sigma$. By induction hypothesis, $plab(q)$ is the longest suffix of t in $Pref(D)$ and therefore $plab(q') = plab(q) \cdot \sigma$ is the longest suffix of $t' = t \cdot \sigma$ in $Pref(D)$.
Case 2: $\delta_D(q, \sigma)$ is not defined. In this case $plab(q) \cdot \sigma$ is not in $Pref(D)$. If $plab(q) = a_1 \dots a_r$, then $q' = \delta(q, \sigma) = \delta^*(q_0, a_2 \dots a_r \sigma)$ and $r \leq |t|$. Using the induction hypothesis for text $a_2 \dots a_r \sigma$ we obtain that $plab(q')$ is the longest suffix of $a_2 \dots a_r \sigma$ in $Pref(D)$. Since $a_1 \dots a_r \sigma$ is not in $Pref(D)$ the longest suffix of t' in $Pref(D)$ is a suffix of $a_2 \dots a_r \sigma$. Hence $plab(q')$ is the longest suffix of t' in $Pref(D)$. □

Corollary 9.1.3 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be as above, let $q \in Q$ and let p' denote the failure state for q. Then $plab(p')$ is the longest proper suffix of $plab(q)$ in $Pref(D)$.*

Proof. As above, let $plab(q) = a_1 \dots a_r$. Obviously $a_2 \cdot \ldots \cdot a_n$ is the longest proper suffix of $a_1 \cdot \ldots \cdot a_n$. Proposition 9.1.2 shows that $plab(p')$ is the longest suffix of $a_2 \cdot \ldots \cdot a_n$ that is in $Pref(D)$. The result follows. □

In particular, Corollary 9.1.3 shows that the failure state p' of a state q always has smaller depth than q.

Proposition 9.1.4 *Let D and $\mathcal{A}$ be as above, let $q \in Q$. Then $q \in F$ iff a suffix of $plab(q)$ is in D.*

Proof. The proof proceeds by induction on $|plab(q)|$. For $|plab(q)| = 0$ the claim is trivially fulfilled.
For the induction step we consider a state q' such that $|plab(q')| = n + 1$. By induction hypothesis we assume that for any state $q \in Q$ such that $|plab(q)| < |plab(q')|$ the claim is fulfilled.
("⇒") If $q' \in F$, then (i) $q' \in F_D$ and $plab(q')$ is in D (trie construction, we are done), or (ii) its failure state p' is in F (cf. Second Step above). In the latter case, since $|plab(p')| < |plab(q')|$ (previous corollary) the induction hypothesis for p' is fulfilled, which shows that a suffix of $plab(p')$ is in D. The previous corollary implies that a suffix of $plab(q')$ is in D.
("⇐") Conversely assume that a suffix of $plab(q')$ is in D. If $plab(q')$ is in D, then $q' \in F_D \subseteq F$. Otherwise a proper suffix β of $plab(q')$ is in D. The previous corollary shows that for the failure state p' of q' we have that $plab(p')$ is the longest proper suffix of $plab(q')$ in $Pref(D)$. Clearly β is a suffix of $plab(p')$. The induction hypothesis shows that $p' \in F$, which implies that $q' \in F$. □

From the above results we obtain

Corollary 9.1.5 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, Q, \delta \rangle$ be the deterministic finite-state automaton constructed by the above method for the dictionary D. Then $L(\mathcal{A}) = \Sigma^* \cdot D$.*

Proof. Let $t \in \Sigma^*$ and $q := \delta^*(q_0, t)$. If $t \in L(\mathcal{A})$, then $q \in F$, by Proposition 9.1.4 a suffix of $plab(q)$ is in D. Since $plab(q)$ is a suffix of t (Proposition 9.1.2), a suffix of t is in D and $t \in \Sigma^* \cdot D$. Conversely, if $t \in \Sigma^* \cdot D$, then a suffix s of t is in D. By Proposition 9.1.2, s is s suffix of $plab(q)$. By Proposition 9.1.4 $q \in F$ and $t \in L(\mathcal{A})$. □

From a procedural point of view the corollary means that when running over a text t with the automaton $\mathcal{A}$ we reach a final state exactly at those points where we have read an infix of t that represents an entry of D.

C(M)-implementation

We first show how to implement the first step, the construction of the trie.

Program 9.1.6 The following program presents an inductive algorithm for building the trie $\mathcal{A}_D$ for a given dictionary D. We assume that D is represented as a finite list of words of the form

$$D = \{\alpha^{(i)} \mid i = 1, \ldots, n\}.$$

Here $\alpha^{(i)} \in \Sigma^+$ and $\alpha^{(i)} \neq \alpha^{(j)}$ for $1 \leq i, j \leq n$, $i \neq j$. The automaton $\mathcal{A}_D$ is constructed in inductive manner. We start with the trivial automaton $\mathcal{A}_D^{(0)}$ with one start state and empty transitions function. In the program, the alphabet Σ is denoted Σ_D.

```
1  import ← "Section_8.1.cm";
2  trieAutomaton : ALPHABET × WORD* → DFSA;
3  trieAutomaton(Σ_D, D) := A_D, where
4     A_D := induction
5        step 0 :
6           A_D^(0) := (Σ_D, {q_0}, q_0, ∅, ∅), where
7              q_0 := 1;
8              ;
```

Each of the following steps adds a new word to the trie. Assume that we have built the trie representing the dictionary $D^{(k)} = \{\alpha^{(i)} \in D \mid i \in \{1, 2, \ldots, k\}\}$ for some $k, 1 \leq k \leq n$. In step $k+1$ we have to add the word $\alpha^{(k+1)}$ to the automaton. Let π denotes the longest sequence of states reachable along a path of $\mathcal{A}_D^{(k)}$ when reading a prefix of $\alpha^{(k+1)}$, starting from q_0. For the function stateseq, see Algorithm 8.1.15.

```
9           step k + 1 :
10             (Σ, Q, q_0, F, δ) := A_D^(k);
11             α := (D)_{k+1};
12             π := stateseq(δ, q_0, α);
```

We introduce a list of new states π' for extending the state sequence π to a sequence fully representing $\alpha^{(k+1)}$. The concatenation $\pi'' = \pi \cdot \pi'$ will be the sequence of the states on the complete path for the full string $\alpha^{(k+1)}$.

```
13          π' := ⟨|Q| + 1, ..., |Q| + |α| − |π| − 1⟩;
14          π'' := π · π';
```

Now we are ready to define $\mathcal{A}_D^{(k+1)}$:

```
15          A_D^(k+1) := (Σ, Q ∪ set(π'), q_0, F ∪ {(π'')_|π''|}, δ'), where
16              δ' := δ ∪ {(((π'')_i, (α)_i), (π'')_{i+1}) | i ∈ {|π|, ..., |π''| − 1}};
17              ;
```

Obviously, $\mathcal{A}_D^{(k+1)}$ represents the dictionary $\{\alpha^{(i)} \in D \mid 1 \leq i \leq k+1\}$. Hence, after n steps

```
18          until k = |D|
19          ;
20      ;
```

we obtain the tree-shaped deterministic finite-state automaton

$$\mathcal{A}_D = \langle \Sigma, Q_D, q_0, F_D, \delta_D \rangle ,$$

which represents D.

Program 9.1.7 Given $\mathcal{A}_D$ we now construct the deterministic finite-state automaton $\mathcal{A}$ representing the language $L(\mathcal{A}) = \Sigma^* \cdot D$.

```
1   import ← "Program_9.1.6.cm";
2   import ← "Program_7.1.4.cm";
3   AhoCorasick : ALPHABET × WORD* → DFSA;
4   AhoCorasick(Σ_D, D) := A, where
5       (Σ, Q, q_0, F_D, δ_D) := trieAutomaton(Σ_D, D);
```

First, we import Program 9.1.6 for building the trie and Program 7.1.4 for sorting the states (Lines 1,2). Recall that $plab(q)$ and $d(q)$ respectively denote the path label and depth of the trie state q. Hence $\delta_D^*(q_0, plab(q)) = q$. In the program we define the corresponding function for the computation of $plab(q)$. A partial order on Q_D is introduced. We say that q' is lower than q'' if $d(q') < d(q'')$.

```
6       plab : STATE → WORD;
7       plab := {(C_δ(δ_D, q_0, (σ)_{1,...,i}), (σ)_{1,...,i}) | σ ∈ D, i ∈ {0, ..., |σ|}};
8       lower : STATE × STATE → IB;
9       lower(q', q'') := |plab(q')| < |plab(q'')| ;
```

Here $C_\delta(\delta_D)$ encodes δ_D^*. By sorting Q_D with respect to the ordering $<$ (lower)

we construct the list $P = \langle p_1, p_2, \ldots, p_h \rangle$, which thus describes a linear ordering for the states of Q_D that respects the partial order. Note that $p_1 = q_0$. This list defines a "breadth-first" visiting order of the states such that states with short path label are visited first.

10 $P := \mathrm{sort}(Q \textbf{ as } \mathcal{STATE}^*, lower);$

In order to define the automaton $\mathcal{A}$ for pattern search based on the given dictionary D we inductively visit all states of $\mathcal{A}_D$ in the above specified breadth-first order.

The automaton $\mathcal{A}^{(1)}$ that results from our visit at q_0 adds a loop transition to q_0 with label σ for all letters where q_0 does not have a σ transition in $\mathcal{A}_D$.

11 $A :=$ **induction**
12 **step** 1 :
13 $A^{(1)} := (\Sigma, Q, q_0, F_D, \delta),$ **where**
14 $\delta := \delta_D \cup \{((q_0, \sigma), q_0) \mid \sigma \in \Sigma \;\&\; \neg(!\,\delta_{\mathrm{D}}(q_0, \sigma))\};$
15 ;

Let us assume that after visiting p_i (where $1 \leq i < h$) we have constructed the automaton $A^{(i)}$. Let $q = p_{i+1}$ denote the next state on our list of states to be visited. Let $plab(q) = \alpha_q = a_1 \ldots a_r$ be the path label for q. For $1 \leq j \leq r$, let $q_j := \delta_D^*(q_0, a_1 \ldots a_j)$.

16 **step** $i + 1$:
17 $(\Sigma, Q, q_0, F', \delta') := A^{(i)};$
18 $q := (P)_{i+1};$
19 $\alpha_q := \mathrm{plab}(q);$
20 $r := |\alpha_q|\,;$

The new automaton $A^{(i+1)}$ is obtained by adding the necessary transitions for q. We first compute the failure state $p' := \delta'^*_i(q_0, a_2 a_3 \ldots a_r)$ for q.

21 $p' := \mathrm{C}_\delta(\delta', q_0, (\alpha_q)_{2,\ldots,r});$

Given the failure state p' we define

22 $A^{(i+1)} := (\Sigma, Q, q_0, F'', \delta''),$ **where**
23 $\delta'' := \delta' \cup \{((q, \sigma), \delta'(p', \sigma)) \mid \sigma \in \Sigma \;\&\; \neg(!\,\delta_{\mathrm{D}}(q, \sigma))\};$
24 $F'' := F' \cup \begin{cases} \{q\} & \textbf{if } p' \in F' \\ \emptyset & \textbf{otherwise;} \end{cases}$
25 ;

Line 23 adds failure transitions from q to $\delta'(p', \sigma)$ with label σ for all letters σ where $\delta_D(q, \sigma)$ is undefined. Line 24 states that q is added as a new member to the list of final states if the failure state p' is final. The automaton $\mathcal{A} = \langle \Sigma, Q, q_0, Q, \delta \rangle$ obtained after the final step h represents the output of our algorithm.

```
26          until i = |P|
27          ;
28      ;
```

9.2 Linear computation of the Aho-Corasick automaton

Program 9.1.7 leads to a time complexity which is quadratic in the worst case. The problem is that the calculation of the failure state p' for a node q requires a traversal of the automaton with the longest proper suffix of $plab(q)$ (see Line 19 of Program 9.1.7). To overcome this problem, during the breadth-first traversal of the trie we maintain a queue. The queue contains pairs of the form $\langle p', q\rangle$ with the following entries:

1. the state q to be processed,
2. the failure state p' for q.

The important point to note is that in our previous algorithm the traversal of the automaton is only used to find the correct failure state p'. However, the pairs in the queue can be calculated in constant time from predecessor pairs! Consider the situation where the pair $\langle p'_{old}, q_{old}\rangle$ has been calculated. Reaching state q_{new} with letter a_r in the trie we want to define the new pair. The above definition of the failure state shows that the new failure state is $p'_{new} = \delta(p'_{old}, a_r)$. It is important to note that the transition from p'_{old} to p'_{new} may be an additional transition added after the trie construction. This shows how maintaining pairs with states and corresponding failure states our earlier traversal of the automaton with a suffix of the path label can be avoided.

Program 9.2.1 The program below constructs the Aho-Corasick automaton for a given dictionary D. In Line 4 the trie associated with the dictionary D is built. L (cf. Line 5) represents a queue with pairs of the form $\langle p', q\rangle$ to be processed. The actual queue L and the actual automaton A are defined by induction (Line 6). Their corresponding initialization in Step 0 is respectively found in Lines 8 and 9-11.

The initial queue $L^{(0)}$ contains the pairs associated with the failure state $p' = q_0$ and the direct successor states $q = \delta_D(q_0, \sigma)$ of q_0 in the trie reached with letters of the alphabet (Line 8). New transitions for q_0 are added (Line 10).

```
1   import ← "Program_9.1.6.cm";
2   AhoCorasick : ALPHABET × WORD* → DFSA;
3   AhoCorasick(Σ_D, D) := A, where
4       (Σ, Q, q_0, F_D, δ_D) := trieAutomaton(Σ_D, D);
5       L ∈ (STATE × STATE)*;
6       (A, L) := induction
7           step 0 :
8               L^(0) := ⟨(q_0, δ_D(q_0, σ)) | σ ∈ Σ & ! δ_D(q_0, σ)⟩;
9               A^(0) := (Σ, Q, q_0, F_D, δ), where
10                  δ := δ_D ∪ {((q_0, σ), q_0) | σ ∈ Σ & ¬(! δ_D(q_0, σ))};
11                  ;
```

Step $i+1$. This step comprises three tasks. The first task (Line 14) is to select from the queue L the next state q to be treated and the corresponding values for the current pair $\langle p', q\rangle$.

The second task is the addition of a new pairs to the queue for each direct successor state q_{new} of q in the trie (Line 15). As explained above the new p' is defined as: $p'_{new} = \delta'(p', \sigma)$.

The third task is the extension of the automaton (Lines 16-19) with the missing information for q, which involves the addition of new transitions (Line 17) and the addition of q to the set of final states in specific cases (Line 18).

```
12        step i + 1 :
13            (Σ, Q, q_0, F', δ') := A^(i);
14            (p', q) := (L^(i))_{i+1};
15            L^(i+1) := L^(i) · ⟨(δ'(p', σ), δ'(q, σ)) | σ ∈ Σ & ! δ_D(q, σ)⟩;
16            A^(i+1) := (Σ, Q, q_0, F'', δ''), where
17                δ'' := δ' ∪ {((q, σ), δ'(p', σ)) | σ ∈ Σ & ¬(! δ_D(q, σ))};
18                F'' := F' ∪ { {q}   if p' ∈ F'
                              { ∅     otherwise;
19                ;
20        until i = |L^(i)|
21        ;
22    ;
```

Clearly the above algorithm implements the improved method described above. Reading the input dictionary takes time proportional to the size of the dictionary. The procedure for building the trie has time complexity proportional to the size of the dictionary. The number of steps required for processing a single state in the main procedure is proportional to the alphabet size. The space requirements for a state and its transitions are proportional to the alphabet size as well. Thus the space and time complexity of the algorithm is $O(\|D\| + m + |\Sigma| \cdot m)$, where m is the number of states in the trie. Obviously $m \leq \|D\| + 1$ thus the time complexity of the algorithm is proportional to the size of the dictionary for a fixed alphabet.

9.3 Space-efficient variant – construction of the Aho-Corasick f-automaton

Depending on the size of the alphabet, the Aho-Corasick automaton obtained from the above construction can become very large: due to the additional transitions added at the second step of the construction, the number of transitions departing from any state is equal to the alphabet size. We now introduce the concept of deterministic finite-state automata with *failure transitions*. It is then shown how to compute for a given dictionary D an automaton with failure transitions recognizing the language $\Sigma^* \cdot D$. The computation of this automaton needs time and space linear in the size of $\|D\|$ and *does not depend on the alphabet size*. As a consequence, in practice depending on the size of the input alphabet the automaton with failure transitions is much smaller than the Aho-

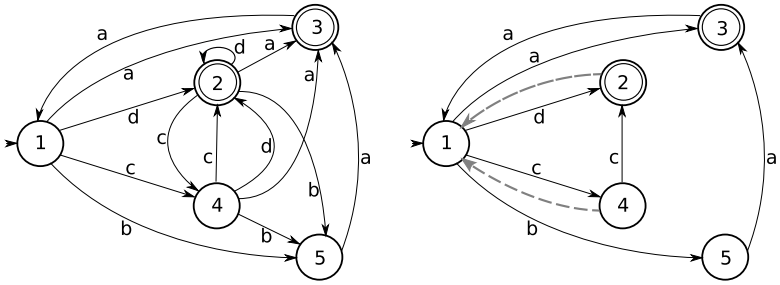

Figure 9.3: Illustration for Example 9.3.1: Compressing deterministic finite-state automata using failure transitions.

Corasick automaton described above. Still, it is shown that processing a text t with the new automaton only needs time linear in the length of t.

Automata with failure transitions

Since finite-state devices often become very large, many compression methods have been investigated [Daciuk and Weiss, 2012, Mehryar Mohri, 2015, P.Pisolkar and Lahane, 2015]. One attractive option is the use of failure transitions. When reading a string, failure transitions are used if the current state reached does not offer a transition for the next input symbol. The basic idea is to follow in this situation a chain of failure transitions until a state is found that offers a transition for the symbol. In this way, transitions can be shared between distinct states, which helps to reduce the numbers of transitions needed and the size of the device. Failure transitions for finite-state automata have been considered, e.g., in [Aho and Corasick, 1975, Mohri, 1995, Crochemore and Hancart, 1997, Kourie et al., 2012, Björklund et al., 2014].

The use of failure transitions provides a significant compression of the resulting finite-state devices at the price of a modest increase of traversal speed under some additional assumptions.

Example 9.3.1 As an illustrating example, consider the deterministic automaton on the left-hand side of Figure 9.3. For State 2 and each symbol $\sigma \in \{a, b, c, d\}$ we have $\delta(2, \sigma) = \delta(1, \sigma)$. For State 4 and each symbol $\sigma \in \{a, b, d\}$ we have $\delta(4, \sigma) = \delta(1, \sigma)$. After introducing failure transitions from 2, and 4 to 1 we can eliminate all standard transitions departing from State 2, and all standard transitions except $\langle 4, c, 2\rangle$ departing from State 4. The new automaton with failure transitions is shown on the right-hand side. Failure transitions are shown in grey using dashed lines. Note that the resulting automaton is much smaller. When reading any symbol σ, say, in state 2, we follow the failure link to 1 and use the σ-link of 1 to find the target state. In this example, some transitions of the original automaton now need 2 steps, which does not have a major influence on processing time.

We first present the general concept.

Definition 9.3.2 A *failure deterministic finite-state automaton* or *f-automaton* is a tuple $\mathcal{F} = \langle \Sigma, Q, q_0, F, \delta, f \rangle$, such that $\langle \Sigma, Q, q_0, F, \delta \rangle$ is a deterministic finite-state automaton and $f : Q \to Q$ is a (partial) function called the *failure transition function.*

In the present context we are only interested in automata with failure transitions of a particular form. The *depth* of a state q of a trimmed deterministic finite-state automaton is the length of the shortest path leading from the initial state to q. It is denoted in the form $d(q)$.

Definition 9.3.3 A *backwards failure deterministic finite-state automata* or *backwards f-automaton* is a trimmed f-automaton $\mathcal{F} = \langle \Sigma, Q, q_0, F, \delta, f \rangle$ such that for every $q \in Q$ such that $f(q)$ is defined we have $d(q) > d(f(q))$.

Definition 9.3.4 Let $\mathcal{F} = \langle \Sigma, Q, q_0, F, \delta, f \rangle$ be a backwards f-automaton. The *completed transition function* $\delta_f : Q \times \Sigma \to Q$ is inductively defined as

$$\delta_f(q, \sigma) := \begin{cases} \delta(q, \sigma) & \text{if } \delta(q, \sigma) \text{ is defined,} \\ \delta_f(f(q), \sigma) & \text{otherwise, if } f(q) \text{ is defined.} \end{cases}$$

Similarly as δ and f also δ_f is a partial function. The *generalized completed transition function* is the (partial) function $\delta_f^* : Q \times \Sigma^* \to Q$ inductively defined as

1. For all $q \in Q$: $\delta_f^*(q, \varepsilon) := q$,
2. For all $q \in Q$, $u \in \Sigma^*$ and $\sigma \in \Sigma$ such that $\delta_f^*(q, u)$ and $\delta_f(\delta_f^*(q, u), \sigma)$ are defined: $\delta_f^*(q, u\sigma) := \delta_f(\delta_f^*(q, u), \sigma)$.

The *language* of the backwards f-automaton $\mathcal{F}$ is defined as

$$L(F) : \{w \in \Sigma^* \mid \delta_f^*(q_0, w) \in F\}.$$

Recall that the notation $\delta_f^*(q_0, w)$ implies that δ_f^* is defined for $\langle q_0, w \rangle$. An alternative definition of the completed transition function is as follows. For each $\sigma \in \Sigma$, let us inductively define the completed failure function $f_\sigma^* : Q \to Q$ as

$$f_\sigma^*(q) := \begin{cases} q & \text{if } \delta(q, \sigma) \text{ is defined,} \\ f_\sigma^*(f(q)) & \text{otherwise, if } f(q) \text{ is defined.} \end{cases}$$

In this case we have

$$\delta_f(q, \sigma) = \delta(f_\sigma^*(q), \sigma).$$

The intuition behind δ_f is that when the automaton encounters the symbol σ in a state q, then it follows the failure transitions until it reaches state $f_\sigma^*(q)$ from which it can consume σ with a transition in δ. But note that a state can be reached that neither has a σ-transition nor a failure link (partiality of δ_f).

When reading a text with a backwards f-automaton the maximal number of failure transition steps cannot exceed the number of standard transitions where we read a text symbol. Failure transitions can only be made when at least the same number of standard transitions have been made before.

Proposition 9.3.5 *Let $\mathcal{F} = \langle \Sigma, Q, q_0, F, \delta, f \rangle$ be a backwards f-automaton. Then the time complexity for recognizing of text of length n by $\mathcal{F}$ is $O(n)$ and does not depend on the size of $\mathcal{F}$.*

Proof. Consider a situation where we have read the i-th symbol of an input text t and are in state p. The number $i - d(p)$ is called the "virtual position" reached at this point. When starting to read t the virtual position is 0. When using a transition $((p, a), q) \in \delta$ the virtual position cannot be decreased. Each failure transition leads to a state with smaller depth, which means that the virtual position is explicitly increased. At any point, the virtual position can only be an integer between 0 and $|t|$. Hence the total number of failure transitions when reading t is bounded by $|t|$. The result follows. □

Computing the Aho-Corasick f-automaton

We now describe a procedure for computing, given the dictionary $D \subseteq \Sigma^*$, a backwards f-automaton recognizing the language $\Sigma^* \cdot D$. With this construction the failure link from a state q simply will lead to the failure state of q in the sense of Section 9.1. The resulting automaton is called the *Aho-Corasick f-automaton*.

The new construction is a simplification of the previous one. Again we proceed in two steps.

In the **first step** we build the trie representing D.

As before, the **second step** is based on a procedure where we visit all states of the trie in a breadth-first manner, starting at the initial state q_0. As in Program 9.2.1 we maintain a queue with entries of the form $\langle p', q \rangle$.

We first complete the initial state q_0 adding a loop transitions for any symbol $\sigma \in \Sigma$ where $\delta_D(q_0, \sigma)$ is undefined. The queue is initialized with all pairs of the form $\langle q_0, q \rangle$ where we have a transition $((q_0, \sigma), q)$ in the trie.

At the later steps we do not add any new (proper) transition to the states $q \neq q_0$, but only extend the failure function f. A state $q \neq q_0$ is visited when we treat a pair $\langle p', q \rangle$ of the queue. We simply define $f(q) := p'$. For maintaining the queue a new pair $\langle p'_{new}, q_{new} \rangle$ is defined for each $\sigma \in \Sigma$ where $\delta_D(q, \sigma)$ is defined. The components are $q_{new} := \delta_D(q, \sigma)$ and $p'_{new} := \delta_f(p', \sigma)$. As defined above, the state $\delta_f(p', \sigma)$ is obtained by following the chain of failure transitions from p' until reaching a state with a transition in δ. The target state of this transition is $\delta_f(p', \sigma)$. As before, if p' is in the extended set of final states, then also q is made final.

Remark 9.3.6 We shall see below that for each pair $\langle p', q \rangle$ in the queue p' is always the failure state of q as defined in Section 9.1. This implies that $d(p') < d(q)$ (cf. Corollary 9.1.3) and p' has been visited before, which ensures that $\delta_f(p', \sigma)$ is defined at this point.

The resulting backwards failure deterministic finite-state automaton for our example dictionary is shown in Figure 9.4. Failure transitions are presented using dashed grey lines. The following proposition shows the correctness of above algorithm for computing the Aho-Corasick f-automaton.

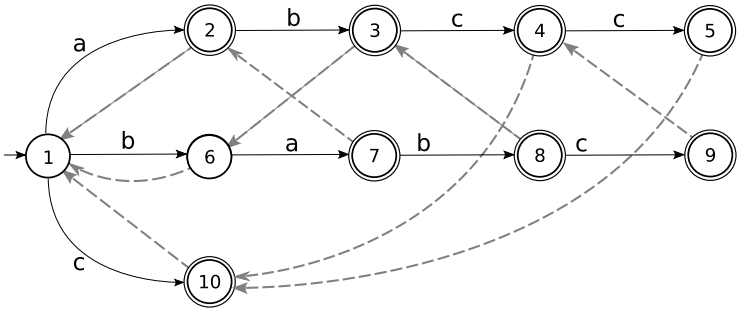

Figure 9.4: The Aho-Corasick f-automaton for the dictionary $\mathcal{D} = \{a, ab, abcc, babc, c\}$.

Proposition 9.3.7 *Let $\mathcal{FA} = \langle \Sigma, Q, q_0, F, \delta, f \rangle$ be the Aho-Corasick backwards failure deterministic finite-state automaton for D built as above. Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta' \rangle$ be the Aho-Corasick deterministic finite-state automaton for D.*

1. *For each state $q_0 \neq q \in Q$ always $f(q)$ is the failure state of q as defined in Section 9.1.*
2. *For each state $q \in Q$ and $\sigma \in \Sigma$ we have $\delta'(q, \sigma) = \delta_f(q, \sigma)$.*

Proof. The two claims are proved simultaneously by induction on the depth $d(q)$ in the trie. By δ_D we denote the transition function of the trie.

For $d(q) = 0$ we have $q = q_0$ and only Claim 2 is relevant. The statement follows from the initialization steps and the definition of δ_f: if $\delta_D(q_0, \sigma)$ is defined, then $\delta'(q_0, \sigma) = \delta_f(q_0, \sigma) = \delta_D(q_0, \sigma)$. In the other case we have $\delta'(q, \sigma) = \delta_f(q, \sigma) = q_0$.

For $d(q) = 1$ the initialization of the queue in the second step shows that $f(q) = q_0$, which is the failure state of q. Statement 1 is trivial. Let $\sigma \in \Sigma$. If $\delta_D(q, \sigma)$ is defined, then $\delta'(q, \sigma) = \delta_D(q, \sigma) = \delta_f(q, \sigma)$. In the other case, $\delta'(q, \sigma) = \delta'(q_0, \sigma) = \delta_f(q_0, \sigma) = \delta_f(f(q), \sigma) = \delta_f(q, \sigma)$, which proves Claim 2.

For $d(q) > 1$ let $q = \delta_D(q_{old}, \tau)$, let $\langle p'_{old}, q_{old} \rangle$ denote the entry of the queue for q_{old}. By induction hypothesis $p'_{old} = f(q_{old})$ is the failure state of q_{old}. When visiting q_{old} the pair $\langle \delta_f(p'_{old}, \tau), q \rangle$ had been added to the queue, which implies that $f(q) = \delta_f(p'_{old}, \tau)$. The induction hypothesis shows that $\delta_f(p'_{old}, \tau) = \delta'(p'_{old}, \tau)$, which is the failure state of q. Statement 1 follows. Let $\sigma \in \Sigma$. If $\delta_D(q, \sigma)$ is defined, then $\delta'(q, \sigma) = \delta_D(q, \sigma) = \delta_f(q, \sigma)$. In the other case, since $f(q)$ is the failure state of q, using the induction hypothesis we obtain $\delta'(q, \sigma) = \delta'(f(q), \sigma) = \delta_f(f(q), \sigma) = \delta_f(q, \sigma)$, which proves Claim 2. □

Remark 9.3.8 In order to achieve linear complexity independent from the alphabet size, the explicit addition of loop transitions for the initial state q_0 in the initialization of the second step has to be avoided. In practice we do not have to add any new transition for q_0 since all these transitions lead back to the initial state. When applying the automaton we have to use a procedure that always goes back to the initial state when reading from q_0 a letter σ where no transition is found.

Program 9.3.9 The following program implements the new procedure.

```
1  import ← "Program_9.1.6.cm";
2  FTRANSITIONS is STATE → STATE;
3  FDFSA is ALPHABET × 2^(STATE) × STATE × 2^(STATE) ×
           DTRANSITIONS × FTRANSITIONS;
4  C_f : (DTRANSITIONS × FTRANSITIONS) × STATE×
           SYMBOL → STATE;
5  C_f((δ, f), q, σ) := p, where
6     p ∈ STATE;
7     p := induction
8        step 0 :
9           p^(0) := q;
10       step i + 1 :
11          p^(i+1) := f(p^(i));
12       until ! δ(p^(i), σ)
13       ;
14    ;
15 δ_f : (DTRANSITIONS × FTRANSITIONS) × STATE×
           SYMBOL → STATE;
16 δ_f((δ, f), q, a) := δ(C_f((δ, f), q, a), a);
17 AhoCorasick : ALPHABET × WORD* → FDFSA;
18 AhoCorasick(Σ, D) := A, where
19    (Σ_D, Q, q_0, F_D, δ_D) := trieAutomaton(Σ, D);
20    L ∈ (STATE × STATE)*;
21    (A, L) := induction
22       step 0 :
23          L^(0) := ⟨(q_0, δ_D(q_0, σ)) | σ ∈ Σ & ! δ_D(q_0, σ)⟩;
24          A^(0) := (Σ, Q, q_0, F_D, δ, ∅), where
25             δ := δ_D ∪ {((q_0, σ), q_0) | σ ∈ Σ & ¬(! δ_D(q_0, σ))};
26             ;
27       step i + 1 :
28          (p', q) := (L^(i))_{i+1};
29          (Σ, Q, q_0, F', δ', f') := A^(i);
30          L^(i+1) := L^(i) · ⟨(δ_f((δ', f'), p', σ), δ_D(q, σ)) | σ ∈ Σ & ! δ_D(q, σ)⟩;
31          A^(i+1) := (Σ, Q, q_0, F'', δ', f' ∪ {(q, p')}), where
32             F'' := F' ∪ { {q}  if p' ∈ F'
                           { ∅    otherwise;
33             ;
34       until i = |L^(i)|
35       ;
36    ;
```

After importing the program for building the trie the program starts with the definition of the types for failure transitions (Line 2) and f-automata (Line 3). In Lines 4-14 we inductively define the values $C_f((\delta, f), q, \sigma)$ which correspond to $f^*_\sigma(q)$. The algorithm is correct for non-cyclic failure transition functions, which is the case for backwards f-automata. Then in Lines 15-16 the function δ_f is implemented. In Lines 17-36 the procedure for constructing the Aho-Corasick

f-automaton is given. For simplicity we here add new loop transitions to the initial state, see Line 25. When we treat a pair with failure state $f(q) = p'$ (see Line 28) and proceed with the new letter σ, adding $\delta_D(q, \sigma)$ to the queue, the corresponding failure state is computed in accordance with the definition above (Line 30).

Remark 9.3.10 The implementation of the Unix command "fgrep" or "grep -F" is based on the Aho-Corasick algorithm presented above. The Aho-Corasick algorithm is also used in the open source tool NCBI BLAST for DNA and protein sequence searching[2].

[2] https://blast.ncbi.nlm.nih.gov/Blast.cgi

Chapter 10

The minimal deterministic finite-state automaton for a finite language

A fundamental task in natural language processing is the efficient representation of lexica. From a computational point, lexica need to be represented in a way directly supporting fast access to entries, and minimizing space requirements. A standard method is to represent lexica in the form of a minimal deterministic (classical) finite-state automaton. To reach such a representation it is of course possible to first build the trie of the lexicon and then to minimize this automaton afterwards. However, in general the intermediate trie is much larger than the resulting minimal automaton. Hence a much better strategy is to use a specialized algorithm to directly compute the minimal deterministic automaton in an incremental way. Procedures for direct construction of the minimal automaton has been presented in [Mihov, 1998a, Daciuk et al., 2000, Mihov and Maurel, 2001, Carrasco and Forcada, 2002, Maurel and Guenthner, 2005]. In this chapter we describe such a procedure.

10.1 Formal construction

When building the minimal deterministic automaton for a lexicon, a desirable method is to add the words one by one, and after each word to obtain an almost minimal automaton. If we want to add a new word v to the automaton, we could try first to find the longest path starting from the initial state with a prefix of v to a state q in the automaton and then to add a chain of new states and transitions from q for recognizing the rest of v. This procedure though has the problem that if there are more paths from the initial state to q, after adding the chain for the rest of v we will add additional (not-desired) words to the language of the automaton. Therefore we need to ensure that the state q is reached by a single path from the starting state. Below we formalize these ideas. We first present some definitions and propositions needed for the construction, following the approach in [Mihov, 1998a].

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

Definition 10.1.1 Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic finite-state automaton. The state $q \in Q$ is *convergent* if there is more than one path from q_0 to q.

Definition 10.1.2 Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a trimmed deterministic finite-state automaton. $\mathcal{A}$ is *minimal except for* $w = a_1 a_2 \dots a_k \in \Sigma^*$ if

1. w is a prefix of a word in $L(\mathcal{A})$. We consider the unique path in $\mathcal{A}$

$$\tau_0 \to^{a_1} \tau_1 \dots \to^{a_k} \tau_k, \text{ where } \tau_0 = q_0.$$

   with sequence of states[1]

$$T_w = \{\tau_0, \tau_1, \dots, \tau_k\}.$$

2. Distinct states in $Q \setminus T_w$ are always non-equivalent.
3. There are no convergent states in T_w.

Condition 3 says that $\tau_0 = q_0$ is not a target state of any transition, and for $i > 0$ the only way to reach τ_i is using the a_i-transition from state τ_{i-1}.

Remark 10.1.3 From an intuitive point of view, the usefulness of the above definition relies on the following. Imagine we are given the list of entries of a dictionary. When we build the automaton representing the dictionary we want to identify equivalent states at the earliest possible stage. This is simplified when we process the entries in alphabetic order (see below). In this case it is possible to immediately identify all states q that do not belong to the path T_w given by the last entry w added, and to have the intermediate automaton built minimal except for w. The reason is that the spectrum of transitions leaving from a state q in the set $Q \setminus T_w$ cannot be extended at later steps, due to the alphabetic order in which entries are processed. When we proceed in this way, immediately identifying these states, $Q \setminus T_w$ represents a "minimized part" where distinct states are never equivalent. When adding a new entry w' in an extension step we obtain a new "latest path" $T_{w'}$. In general, several states of the former "latest path" T_w are not in $T_{w'}$. These states now need to be added to the minimized part, testing equivalence with states in the existing minimized part. At this point we have to compare two states where all transitions lead to states in the minimized part. Equivalence testing (cf. Proposition 3.4.18) is simplified since equivalence of target states boils down to identity (cf. Prop. 10.1.6 below). This description is just meant to support intuitions. All formal details are described below.

In what follows, $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ always denotes a trimmed deterministic finite-state automaton.

Proposition 10.1.4 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \dots a_k \in \Sigma^+$, let $T_w = \{\tau_0, \tau_1, \dots, \tau_k\}$ denote the sequence of states for w. Assume that no state in $Q \setminus T_w$ is equivalent to τ_k. Then $\mathcal{A}$ is minimal except for $w' = a_1 a_2 \dots a_{k-1}$.*

[1] Strictly speaking T_w is a finite set of states, which elements are indexed.

Proof. In this case also distinct states in $Q \setminus \{\tau_0, \tau_1, \ldots, \tau_{k-1}\}$ are always non-equivalent. The validity of the above proposition follows directly from Definition 10.1.2. □

Proposition 10.1.5 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \ldots a_k \in \Sigma^+$, let $T_w = \{\tau_0, \tau_1, \ldots, \tau_k\}$ denote the sequence of states for w. Assume that τ_k is equivalent to a state $r \in Q \setminus T$. Then the deterministic finite-state automaton*

$$\mathcal{A}' = \langle \Sigma, Q \setminus \{\tau_k\}, q_0, F \setminus \{\tau_k\}, \delta' \rangle$$

where $\delta' = \delta|_{Q\setminus\{\tau_k\}\times\Sigma} \setminus \{\langle \tau_{k-1}, a_k, \tau_k \rangle\} \cup \{\langle \tau_{k-1}, a_k, r \rangle\}$ is minimal except for $w' = a_1 a_2 \ldots a_{k-1}$ and $L(\mathcal{A}) = L(\mathcal{A}')$.

Proof. Redirecting a transition to an equivalent state does not change the language of the automaton (Proposition 2.1.17). In $\mathcal{A}$, state t_k can only be reached with the transition $\langle \tau_{k-1}, a_k, \tau_k \rangle$. After redirecting this transition, τ_k is not reachable, removing it does not change the language of the automaton as well. Clearly, $\mathcal{A}'$ is trimmed. Since the conditions of Definition 10.1.2 hold, $\mathcal{A}'$ is minimal except for w'. □

Proposition 10.1.6 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \ldots a_k \in \Sigma^*$, let $T_w = \{\tau_0, \tau_1, \ldots, \tau_k\}$ denote the sequence of states for w. Then τ_k is equivalent to $r \in Q \setminus T_w$ iff*

1. *$\tau_k \in F$ iff $r \in F$,*
2. *$\forall \sigma \in \Sigma$: both $\delta(\tau_k, \sigma), \delta(r, \sigma)$ are undefined or $\delta(\tau_k, \sigma) = \delta(r, \sigma)$.*

Proof. The proposition follows from Proposition 3.4.18 and the fact that all transitions from τ_k and r lead to states in $Q \setminus T_w$ and there are no equivalent states in $Q \setminus T_w$. □

Proposition 10.1.7 *Let $\mathcal{A}$ be minimal except for ε and $\mathcal{A}$ be acyclic. Then $\mathcal{A}$ is a minimal deterministic finite-state automaton.*

Proof. Since the automaton is trimmed and acyclic there is no state equivalent to the initial state and therefore there are no equivalent states in the automaton. Hence the automaton $\mathcal{A}$ is minimal (Proposition 3.4.17). □

The next proposition adresses the situation where $\mathcal{A}$ is minimal except for w and we add an extension $v = w a_{k+1} \ldots a_m$ of w.

Proposition 10.1.8 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \ldots a_k \in \Sigma^*$, let $T_w = \{\tau_0, \tau_1, \ldots, \tau_k\}$ denote the sequence of states for w. Let*

$$v = a_1 a_2 \ldots a_k a_{k+1} \ldots a_m \in \Sigma^*$$

be a word such that $\max_{u \in L(\mathcal{A})} u \wedge v = w$. Let

$$\mathcal{A}' := \langle \Sigma, Q \cup T', q_0, F \cup \{\tau_m\}, \delta' \rangle$$

where

- $T' = \{\tau_{k+1}, \ldots, \tau_m\}$ *is a set of new states* ($T' \cap Q = \emptyset$),
- $\delta' = \delta \cup \{\langle \tau_{i-1}, a_i, \tau_i \rangle \mid i = k+1, \ldots, m\}$,

Then $\mathcal{A}'$ *is a deterministic finite-state automaton which is minimal except for* v *and* $L(\mathcal{A}') = L(\mathcal{A}) \cup \{v\}$.

Proof. Since w is the longest common prefix of v and any word u in $L(\mathcal{A})$, the transition $\delta(\tau_k, a_{k+1})$ is not defined and therefore δ' is a function and $\mathcal{A}'$ is a deterministic finite-state automaton. Adding the state sequence T' - the last state τ_m being a final state - is adding v to the language of the automaton. Since the conditions in Definition 10.1.2 hold $\mathcal{A}'$ is a deterministic finite-state automaton minimal except for v. □

The following proposition helps to efficiently find the largest common prefix of a new entry with the set of all earlier entries in case of adding the entries in lexicographical order.

Proposition 10.1.9 *Let* $L \subseteq \Sigma^*$ *be a finite language and let* $u \in L$ *be the lexicographically largest word in* L. *Let* $v \in \Sigma^*$ *be a word that is lexicographically larger than* u. *Then*

$$\max_{u' \in L} u' \wedge v = u \wedge v.$$

Proof. Clearly $u \wedge v$ is a prefix of $\max_{u' \in L} u' \wedge v$. If we assume that for a word $u' \in L$ we have that $|u' \wedge v| > |u \wedge v|$ we will have that u' is lexicographically larger than u which yields a contradiction. □

Adding the entries of the dictionary in alphabetic order. For a given dictionary $D = \{\alpha^{(i)} \mid i = 1, \ldots, |D|\}$ such that $\alpha^{(i)}$ lexicographically precedes $\alpha^{(i+1)}$ for each $i < |D|$, the construction proceeds inductively in the following way:

1. For the first entry $\alpha^{(1)} = a_1 a_2 \ldots a_k$ we construct the trivial automaton $\mathcal{A}^{(1)} = \langle \Sigma, T, \tau_0, \{\tau_k\}, \delta \rangle$, where

   $$\begin{aligned} T &:= \{\tau_0, \tau_1, \ldots, \tau_k\} \text{ is a set of } k+1 \text{ new states,} \\ \delta &:= \{\langle \tau_{i-1}, a_i, \tau_i \rangle \mid i = 1, \ldots, k\}. \end{aligned}$$

   This trimmed deterministic finite-state automaton is minimal except for $\alpha^{(1)}$ and represents the language $\{\alpha^{(1)}\}$.

2. Let $\mathcal{A}^{(n)}$ be the automaton that is minimal except for $\alpha^{(n)}$ and represents the language $\{\alpha^{(i)} \mid i = 1, \ldots, n\}$ where $n < |D|$.

3. As a pre-step for adding the next entry $\alpha^{(n+1)}$ we treat those states τ_i of the state sequence $T_{\alpha^{(n)}}$ where $plab(\tau_i)$ is *not* a prefix of $\alpha^{(n+1)}$. In other words, we treat the states of $T_{\alpha^{(n)}}$ that do not belong to the future path $T_{\alpha^{(n+1)}}$. We start with the last state $\tau_{|\alpha^{(n)}|}$ and proceed backwards. We apply Propositions 10.1.4 and 10.1.5 (depending on the (non-) existence of a state equivalent to the state to be treated) in an iterative manner until we obtain the automaton $\mathcal{A}' = \langle \Sigma, Q, q_0, F, \delta \rangle$ that is minimal except for $w = \alpha^{(n)} \wedge \alpha^{(n+1)}$. Equivalence checking is performed using Proposition 10.1.6. See Figure 10.1 for an illustration.

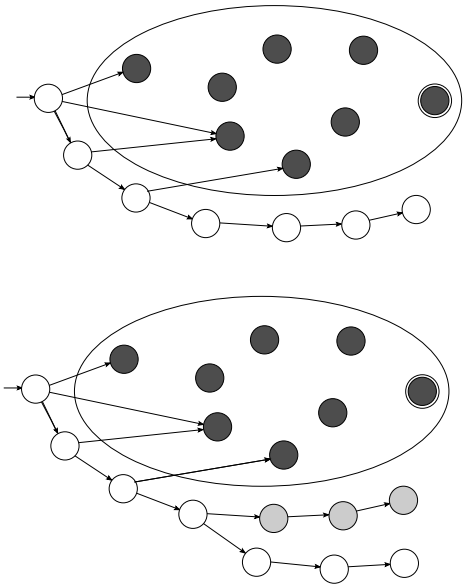

Figure 10.1: Adding a new entry of the dictionary to the automaton. Lexicon entries are treated in lexicographic order. The upper part shows the situation where we have added the first n entries of the dictionary. Dark grey states in the ellipse belong to the minimized part $Q \setminus T_{\alpha^{(n)}}$, white states represent the sequence of states for the n-th entry $\alpha^{(n)}$. When adding the next word, as a pre-step the bright grey states are "minimized", checking equivalence with states in the minimized part. Minimization starts with the right-most bright grey state and proceeds in right-to-left direction. The bright grey state considered at a step is either identified with a dark grey state (bright grey state erased, incoming transitions redirected), or it becomes a new dark grey state.

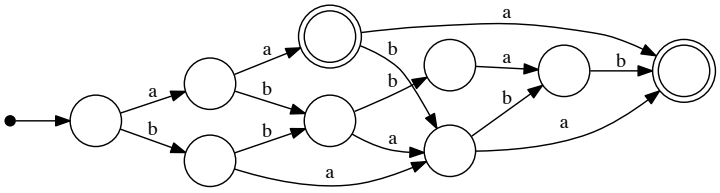

Figure 10.2: The minimal deterministic finite-state automaton for the dictionary with the entries *aa*, *aaa*, *aaba*, *aabbb*, *abaa*, *ababb*, *abbab*, *baa*, *babb*, *bbaa*, *bbabb*, and *bbbab*.

4. From Proposition 10.1.9 we know that $w = \max_{u' \in L(\mathcal{A}^{(n)})} u' \wedge \alpha^{(n+1)}$. Applying Proposition 10.1.8 we construct the automaton $\mathcal{A}^{(n+1)}$ that is minimal except for $\alpha^{(n+1)}$ and represents the language $\{\alpha^{(i)} \mid i = 1, \ldots, n+1\}$.

5. Until the set D is exhausted we increase n by one and loop through points 2-4.

6. Finally we apply Propositions 10.1.4 and 10.1.5 until we construct the automaton $\mathcal{T}$ that is minimal except for ε and represents D. From Proposition 10.1.7 we know that $\mathcal{A}$ is minimal.

As an illustration, Figure 10.2 presents the minimal deterministic finite-state automaton obtained for the dictionary

$$D = \{aa, aaa, aaba, aabbb, abaa, ababb, abbab, baa, babb, bbaa, bbabb, bbbab\}.$$

10.2 *C(M)* implementation of the construction - first version

Program 10.2.1 We now give a *C(M)* implementation that exactly follows this mathematical description. The following function finds the set of states in a set Q that are equivalent to a given state p in accordance with Proposition 10.1.6. Clearly, when using for Q the minimized part, the resulting set of states is either empty or a singleton.

```
1  import ← "Section_8.1.cm";
2  findEquivalent : DFSA × STATE → 2^STATE;
3  findEquivalent((Σ, Q, q0, F, δ), t) := E, where
4      equiv : STATE × STATE → IB;
5      equiv(q, t) := ((q ∈ F) ↔ (t ∈ F)) ∧ ∀c ∈ Σ : ((! δ(q, c) ↔! δ(t, c)) ∧
   (! δ(q, c) → (δ(q, c) = δ(t, c))));
6      E := {q | q ∈ Q & equiv(q, t) ∧ (q ≠ t)};
7      ;
```

The following program lines describe the subprocedure that is used for minimizing the end of a "former latest path". When we call the subprocedure, α presents the former latest word added. For a given automaton $\mathcal{A}$ minimal except for α the function reduceExcept presented below returns the automaton $\mathcal{A}'$ that is minimal except for the prefix $(\alpha)_{1..l}$ of α (here $0 \leq l \leq |\alpha|$ is an input parameter) and is equivalent to $\mathcal{A}$. Applying the function stateseq (cf. Algorithm 8.1.15) we obtain the longest sequence of states τ in $\mathcal{A}$ that can be reached from q_0 reading a prefix of α (Lines 10-12). In the current situation the full word α can be read. Using an induction (Lines 13-27) we "minimize" the states $(\tau)_{l+2}$, $(\tau)_{l+3}$, ..., $(\tau)_{|\alpha|+1}$. This partial minimization starts with $(\tau)_{|\alpha|+1}$ and inductively proceeds backwards until reaching $(\tau)_{l+2}$. After the i-th step, an intermediate automaton $\mathcal{A}'^{(i)}$ is obtained. Let $\mathcal{A}'^{(0)} := \mathcal{A}$ (Line 15). Assume we have constructed $\mathcal{A}'^{(i)}$ for some i $(1 \leq i < |\tau| - l - 1)$. At step $i+1$ we proceed by analyzing state $(\tau)_p$ for $p = |\tau| - i$. We distinguish two cases:

Case 1. In $\mathcal{A}'^{(i)}$ there is no other state equivalent to $(\tau)_p$ (Line 20). In this case we make no changes to the automaton (cf. Propositions 10.1.4).

Case 2. In $\mathcal{A}'^{(k)}$ there is a state $r \neq (\tau)_p$ equivalent to $(\tau)_p$ (Lines 21-24). Here the state $(\tau)_p$ and all outgoing transition from $(\tau)_p$ are removed and the only transition to $(\tau)_p$ is redirected to the equivalent state r. Note that the language of the automaton is not affected (cf. Proposition 10.1.5).

```
8   reduceExcept : DFSA × WORD × ℕ → DFSA;
9   reduceExcept(A, α, l) := A', where
10      τ := stateseq(δ, q0, α), where
11          (Σ, Q, q0, F, δ) := A;
12          ;
13      A' := induction
14          step 0 :
15              A'^(0) := A;
16          step i + 1 :
17              p := |τ| − i;
18              R := findEquivalent(A'^(i), (τ)_p);
19              A'^(i+1) :=
20                  case R = ∅ : A'^(i)
21                  otherwise : (Σ', Q' \ {(τ)_p}, q0', F' \ {(τ)_p}, δ''), where
22                      (Σ', Q', q0', F', δ') := A'^(i);
23                      r := elementOf(R);
24                      δ'' := (δ'
                        \ {(((τ)_p, σ), δ'((τ)_p, σ)) | σ ∈ Σ' & ! δ'((τ)_p, σ)})
                        \ {(((τ)_{p−1}, (α)_{p−1}), (τ)_p)} ∪ {(((τ)_{p−1}, (α)_{p−1}), r)};
25                  ;
26          until |τ| − i = l + 1
27          ;
28      ;
```

Once we have the above minimization subprocedure we now turn to the main procedure. We assume that the dictionary D is represented as a finite list of words of the form

$$D = [\alpha^{(i)} \mid i = 1, \ldots, |D|].$$

Entries are nonempty, distinct, and sorted in lexicographic order. The entries are processed one after one. After each step we obtain an intermediate automaton which is minimal except for the last word added. As our *initialization* we start with the automaton $\mathcal{A}^{(1)}$ for the first word $\alpha^{(1)} = \alpha = \alpha_1\alpha_2 \ldots \alpha_{|\alpha^{(1)}|}$.

```
29  mdic : ALPHABET × WORD* → DFSA;
30  mdic(Σ_D, D) := reduceExcept(A, (D)_|D|, 0), where
31      A ∈ DFSA;
32      A := induction
33          step 1 :
34              α := (D)_1;
35              τ := ⟨1, ..., |α| + 1⟩;
36              A^(1) := (Σ_D, set(τ), (τ)_1, {(τ)_{|α|+1}}, δ), where
```

37 $\delta := \{(((\tau)_i, (\alpha)_i), (\tau)_{i+1}) \mid i \in \{1, \ldots, |\alpha|\}\};$
38 ;

Here τ is the sequence of states for the traversal of α (we use natural numbers for representing the states). The start state of $\mathcal{A}^{(1)}$ is $(\tau)_1$ and $(\tau)_{|\alpha|+1}$ is the only final state. Transitions in δ lead from $(\tau)_i$ with symbol $(\alpha)_i$ to $(\tau)_{i+1}$.

Iteration. Each of the following steps adds a new word to the automaton (Proposition 10.1.8) and is preceded by a partial minimization using the function reduceExcept. Let us assume that we have constructed the automaton $\mathcal{A}^{(n)}$ minimal except for $\alpha^{(n)} = u$ (Line 41) and representing the dictionary $D^n = \{\alpha^{(i)} \mid i \in \{1, 2, \ldots, n\}\}$ for some $n, 1 \leq n < |D|$. We next have to add the word $\alpha^{(n+1)} = v = (D)_{n+1}$ (Line 42) to the automaton. Then the longest sequence of states in $\mathcal{A}^{(n)}$ defined by input v is $\tau := \langle(\tau)_1, (\tau)_2, \ldots, (\tau)_{k+1}\rangle$ where $(\tau)_1 = q_0$ and $(\tau)_{i+1} = \delta((\tau)_i, (v)_i)$ for $i = 1, \ldots, k$. Here k denotes the length of the maximal common prefix of entries u and v (see Lines 42-45).

39 **step** $n + 1$:
40 $v := (D)_{n+1}$;
41 $u := (D)_n$;
42 $\tau :=$ stateseq(δ, q_0, v), **where**
43 $(\Sigma, Q, q_0, F, \delta) := A^{(n)}$;
44 ;
45 $k := |\tau| - 1$;
46 $(\Sigma', Q', q_0', F', \delta') :=$ reduceExcept$(A^{(n)}, u, k)$;

Note that $(\tau)_{k+1}$ represents the last state reached with the common prefix $v_{1..k}$. Applying the function reduceExcept in Line 46 we minimize the states $(\tau)_{k+2}$, $(\tau)_{k+3}$, ..., $(\tau)_{|u|+1}$. The automaton reached is minimal except for $u_{1..k} = u \wedge v$.

We finish the construction of $\mathcal{A}^{(n+1)}$ by introducing (Line 47-51) a sequence of new states and transitions for the remaining suffix of v (cf. Proposition 10.1.8). Numbers $|Q| + 1, \ldots, |Q| + |v| - k$ serve as new states. Our construction guarantees that these numbers do not represent states of $\mathcal{A}^{(n)}$.

47 $T' := \langle|Q'| + 1, \ldots, |Q'| + |v| - k\rangle$;
48 $\tau' := \tau \cdot T'$;
49 $A^{(n+1)} := (\Sigma', Q' \cup \text{set}(T'), q_0', F' \cup \{(T')_{|T'|}\}, \delta'')$, **where**
50 $\delta'' := \delta' \cup \{(((\tau')_i, (v)_i), (\tau')_{i+1}) \mid i \in \{k + 1, \ldots, |v|\}\}$;
51 ;
52 **until** $n = |D|$
53 ;
54 ;

Now $\mathcal{A}^{(n+1)}$ represents the subdictionary $\{\alpha^{(i)} \in D \mid 1 \leq i \leq n + 1\}$. Hence, after $|D|$ steps we receive a partially minimized automaton $\mathcal{A}^{(|D|)}$ representing the full dictionary D that is minimal except for the last word in D.

It remains to complete the minimization. Let τ be the sequence of states for the last input word in D. As a final step, the states $(\tau)_{|\tau|}, \ldots, (\tau)_2$ are minimized applying the function reduceExcept in Line 29. The resulting automaton is the

minimal automaton representing the dictionary D (cf. Proposition 10.1.7).

10.3 Efficient construction of the minimal dictionary automaton

In the search procedure for an equivalent state (function findEquivalent) we actually scan the complete set of all states (Line 6 in Program 10.2.1). As a consequence, the time complexity of the above algorithm is quadratic in the number of states of the resulting automaton. To overcome this problem we now characterize each state in terms of a "signature". This signature contains all information needed for equivalence testing, details are explained below. We then introduce a function that - given the signature of a state - directly returns the equivalent state if such a state exists. The resulting new algorithm is presented below. Using an appropriate structure (e.g. balanced tree, hash,...) on the state signatures the search for an equivalent state can now be performed in time $O(\log(m))$ where m is the number of states in the resulting minimal automaton. Therefore the time complexity of the algorithm presented below is $O(n \log(m))$, where n is the size of the dictionary (see [Mihov, 1998b]).

Program 10.3.1 The new program in Line 2 introduces "state signatures". The signature of a state p is a pair. The first component says if p is a final state (Boolean value). The second component characterizes the outgoing transitions of p. Signatures are only computed for states where the spectrum of transitions is not modified later, and all transitions lead to states that are already "minimized". In this way, once we have the signature of a state, it is not subject to later modifications. The function calcSignature calculates the signature for a given state q in the automaton $(\Sigma, Q, q_0, F, \delta)$ (Lines 3-4).

```
1  import ← "Section_8.1.cm";
2  STSIGN is IB × 2^(SYMBOL×STATE);
3  calcSignature : DFSA × STATE → STSIGN;
4  calcSignature((Σ, Q, q0, F, δ), q) := (q ∈ F, {(c, δ(q, c)) | c ∈ Σ & ! δ(q, c)});
```

In the new program along with the automaton $\mathcal{A}$ we maintain a function h that maps the signatures of minimized states to the corresponding states. Here it is essential that two distinct minimized states always have a distinct signature. As before, for a given automaton $\mathcal{A}$ minimal except for α the function reduceExcept presented below returns the automaton $\mathcal{A}'$ that is minimal except for $(\alpha)_{1..l}$ $(l \leq |\alpha|)$ and is equivalent to $\mathcal{A}$. But in addition the new function takes as an additional argument the mapping h and returns the new mapping h'. Here for equivalence testing the function h is used (Line 18), which gives the gain in efficiency compared to the earlier algorithm. The mapping h is updated, adding the signatures of the new minimized states to the domain (Line 18).

```
5  reduceExcept : DFSA × WORD × IN × (STSIGN → STATE) →
   DFSA × (STSIGN → STATE);
6  reduceExcept(A, α, l, h) := (A', h'), where
7      τ := stateseq(δ, q0, α), where
```

```
8           (Σ, Q, q_0, F, δ) := A;
9           ;
10      (A', h') := induction
11          step 0 :
12              A'^(0) := A;
13              h'^(0) := h;
14          step i + 1 :
15              p := |τ| − i;
16              h_q := calcSignature(A'^(i), (τ)_p);
17              (A'^(i+1), h'^(i+1)) :=
18                  case ¬(! h'^(i)(h_q)) : (A'^(i), h'^(i) ∪ {(h_q, (τ)_p)})
19                  otherwise :
    ((Σ', Q' \ {(τ)_p}, q'_0, F' \ {(τ)_p}, δ''), h'^(i)), where
20                      (Σ', Q', q'_0, F', δ') := A'^(i);
21                      r := h'^(i)(h_q);
22                      δ'' := (δ'
                        \ {(((τ)_p, σ), δ'((τ)_p, σ)) | σ ∈ Σ' & ! δ'((τ)_p, σ)})
                        \ {(((τ)_{p−1}, (α)_{p−1}), (τ)_p)} ∪ {(((τ)_{p−1}, (α)_{p−1}), r)};
23                  ;
24          until |τ| − i = l + 1
25          ;
26      ;
```

The computation of the minimal automaton $\mathcal{A}$ proceeds similarly as in Program 10.2.1. The only new point is the maintenance of the mapping h. In Line 34 the mapping $h^{(1)}$ is initialized. Afterwards the mapping is updated applying the function reduceExcept in Line 44.

```
27  mdic : ALPHABET × WORD* → DFSA;
28  mdic(Σ_D, D) := Proj_1(reduceExcept(A, (D)_{|D|}, 1, h)), where
29      (A, h) ∈ DFSA × (STSIGN → STATE);
30      (A, h) := induction
31          step 1 :
32              α := (D)_1;
33              τ := ⟨1, ..., |α| + 1⟩;
34              (A^(1), h^(1)) := ((Σ_D, set(τ), (τ)_1, {(τ)_{|α|+1}}, δ), ∅), where
35                  δ := {(((τ)_i, (α)_i), (τ)_{i+1}) | i ∈ {1, ..., |α|}};
36                  ;
37          step n + 1 :
38              v := (D)_{n+1};
39              u := (D)_n;
40              τ := stateseq(δ, q_0, v), where
41                  (Σ, Q, q_0, F, δ) := A^(n);
42                  ;
43              k := |τ| − 1;
44              ((Σ', Q', q'_0, F', δ'), h^(n+1)) := reduceExcept(A^(n), u, k, h^(n));
45              T' := ⟨|Q'| + 1, ..., |Q'| + |v| − k⟩;
46              τ' := τ · T';
47              A^(n+1) := (Σ', Q' ∪ set(T'), q'_0, F' ∪ {(T')_{|T'|}}, δ''), where
```

```
48                  δ'' := δ' ∪ {(((τ')_i, (v)_i), (τ')_{i+1}) | i ∈ ⟨k + 1, ..., |v|⟩};
49                  ;
50              until n = |D|
51              ;
52          ;
```

As in the previous algorithm, as a final step the states on the path for the last entry of the dictionary need to be minimized (Line 28).

10.4 Adapting the language of minimal dictionary automata

We now present an efficient method for adding and deleting words from the language of a minimal dictionary automaton, preserving minimality of the automaton [Daciuk et al., 2000]. In this context as a technical preparation we have to consider a situation where an automaton is minimal except for a string $a_1 \cdots a_k$ and we want to make it minimal for an extension $a_1 \cdots a_k, a_{k+1}$. In other words, we need to be able to extend the word except for which the automaton is minimal. The following propositions formalize this concept.

Proposition 10.4.1 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \ldots a_k \in \Sigma^*$, let $T_w = \{\tau_0, \tau_1, \ldots, \tau_k\}$ denote the sequence of states for w. If the state $q = \delta(\tau_k, a_{k+1})$ is not convergent, then $\mathcal{A}$ is minimal except for $w' = a_1 a_2 \ldots a_k a_{k+1}$.*

The above proposition follows directly from Definition 10.1.2 taking into account that the state q is not convergent. In the case where q is convergent, in order to extend the word except for which the automaton is minimal we have to add a new state equivalent to q. Below we formalize this idea.

Definition 10.4.2 Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic finite-state automaton and $\langle p, a, q \rangle$ be a transition in δ. Then the deterministic finite-state automaton $\mathcal{A}'$ is said to be obtained from $\mathcal{A}$ by *cloning state q for the transition* $\langle p, a, q \rangle$ if $\mathcal{A}' = \langle \Sigma, Q \cup \{q'\}, q_0, F', \delta' \rangle$, where

- q' is a new state,
- $\delta' = (\delta \setminus \langle p, a, q \rangle) \cup \langle p, a, q' \rangle \cup \{\langle q', \sigma, \delta(q, \sigma) \rangle \mid \sigma \in \Sigma, !\delta(q, \sigma)\}$,
- $F' = F \cup \begin{cases} \{q'\} & \text{if } q \in F \\ \emptyset & \text{otherwise.} \end{cases}$

Proposition 10.4.3 *Let the deterministic finite-state automaton $\mathcal{A}'$ be obtained from $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ by cloning state q for transition $\langle p, a, q \rangle \in \delta$. Then $\mathcal{A}'$ is equivalent to $\mathcal{A}$.*

The above proposition follows from Proposition 2.1.17.

Proposition 10.4.4 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \ldots a_k \in \Sigma^*$, let $T_w = \{\tau_0, \tau_1, \ldots, \tau_k\}$ denote the sequence of states for w. Let $q = \delta(\tau_k, a_{k+1})$*

and q be convergent. Then the deterministic finite-state automaton $\mathcal{A}'$ obtained from $\mathcal{A}$ by cloning the state q for the transition $\langle \tau_k, a_{k+1}, q \rangle \in \delta$ is minimal except for $w' = a_1 a_2 \dots a_k a_{k+1}$.

Adding a new word to the language of a minimal dictionary automaton. For a given minimal deterministic automaton $\mathcal{A}$ representing the dictionary D the construction for adding a new word proceeds in the following way:

1. As a pre-step for adding the new word v we first obtain the automaton minimal except for $w := v \wedge L(\mathcal{A})$ by cloning all convergent states reached with w. We start from the initial state and apply Propositions 10.4.1 and 10.4.4 (depending on whether the state $q = \delta(\tau_k, a_{k+1})$ is convergent or not) in an iterated manner until we obtain the automaton $\mathcal{A}' = \langle \Sigma, Q', q_0, F', \delta' \rangle$ that is minimal except for w. We remark that cloning a state implies that the successor state on the path for w becomes convergent and has to be cloned as well.
2. Applying Proposition 10.1.8 we construct the automaton $\mathcal{A}''$ that is minimal except for v and represents the language $L(\mathcal{A}) \cup \{v\}$.
3. Finally we apply Propositions 10.1.4 and 10.1.5 until we construct the automaton $\mathcal{A}'''$ that is minimal except for ε and represents $L(\mathcal{A}) \cup \{v\}$. From Proposition 10.1.7 we know that $\mathcal{A}'''$ is minimal.

The following propositions yield the background for the procedure where we delete a word from the automaton.

Proposition 10.4.5 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \dots a_n \in \Sigma^+$, let $T_w = \{\tau_0, \tau_1, \dots, \tau_n\}$ denote the sequence of states for w and let $w \in L(\mathcal{A})$. Let there be at least one outgoing transition from τ_n. Then the automaton $\mathcal{A}' = \langle \Sigma, Q, q_0, F \setminus \{\tau_n\}, \delta \rangle$ is minimal except for w and $L(\mathcal{A}') = L(\mathcal{A}) \setminus \{w\}$.*

The proof follows from the fact that there are no convergent states on the path for q. If however the state τ_n has no outgoing transition, then by making it non-final there is no final state reachable from τ_n (and some of its predecessors) and thus the automaton is not trimmed. The next proposition formalizes this observation.

Proposition 10.4.6 *Let $\mathcal{A}$ be minimal except for $w = a_1 a_2 \dots a_n \in \Sigma^+$, let $w \in L(\mathcal{A})$ and let $T_w = \{\tau_0, \tau_1, \dots, \tau_n\}$ denote the sequence of states for w. Assume that there is no outgoing transition from τ_n and let $|L(\mathcal{A})| > 1$. Let τ_m be the last state in the sequence $\tau_0, \tau_1, \dots, \tau_n$ such that there exists more than one path from τ_m to a final state. Then the automaton*

$$\mathcal{A}' = \left\langle \Sigma, Q \setminus T', q_0, F \setminus \{\tau_n\}, \delta|_{(Q \setminus T') \times \Sigma} \right\rangle,$$

where $T' = \{\tau_{m+1}, \dots, \tau_n\}$ is minimal except for $w' = a_1 a_2 \dots a_m$ and $L(\mathcal{A}') = L(\mathcal{A}) \setminus \{w\}$.

Note that there is more than one path from state τ_m to a final state iff τ_m is final or has more than one outgoing transition. Hence all states in $\{\tau_{m+1}, \dots, \tau_{n-1}\}$ are non-final and have exactly one outgoing transition.

Deleting a word from the language of a minimal dictionary automaton. For a given minimal deterministic dictionary automaton $\mathcal{A}$ representing the dictionary D the construction for deleting a single entry proceeds in the following way:

1. As a pre-step for deleting the word $v = a_1 \dots a_n$ we first build the automaton minimal except for v. We need to have a path for v without convergent states. To this end we start from the initial state and apply Propositions 10.4.1 and 10.4.4 (depending on whether the state $q = \delta(\tau_k, a_{k+1})$ is convergent or not) in an iterative manner until we obtain the automaton $\mathcal{A}' = \langle \Sigma, Q', q_0, F', \delta' \rangle$ that is minimal except for v.

2. Let $\tau_0, \tau_1, \dots, \tau_n$ be the sequence of states in $\mathcal{A}'$ on the path with label v. If there are no outgoing transitions from τ_n we obtain the automaton $\mathcal{A}''$ by deleting the states $\tau_n, \tau_{n-1}, \dots, \tau_{m+1}$, where τ_m is the first state in the backward sequence that either is final or has more than one outgoing transition. State τ_n is removed from the set of final states. According to Proposition 10.4.6 $\mathcal{A}''$ is minimal except for $v_{1,\dots,m}$ and represents the language $L(\mathcal{A}) \setminus \{v\}$. If there are outgoing transitions from τ_n, then $\mathcal{A}''$ is obtained from $\mathcal{A}'$ by removing the state τ_n from the set of final states. By Proposition 10.4.5 $\mathcal{A}''$ is minimal except for v and represents the language $L(\mathcal{A}) \setminus \{v\}$.

3. Finally we apply Propositions 10.1.4 and 10.1.5 until we obtain the automaton $\mathcal{A}'''$ that is minimal except for ε and represents $L(\mathcal{A}) \setminus \{v\}$. From Proposition 10.1.7 we know that $\mathcal{A}'''$ is minimal.

Remark 10.4.7 Recall that our automaton programs are based on the general assumption that states are numberered in the form $1, \dots, |Q|$. For adding a word to the automaton or deleting a word of the automaton we apply the steps described in Propositions 10.4.6 or 10.1.5. Both prodedures delete states from the automaton. At this point the general assumption can get violated. The next program does not maintain the general assumption. If needed one can remap the states using a simple procedure similar to the one in Program 8.1.6.

Program 10.4.8 The following program implements addition and deletion of words in a minimal dictionary automaton while preserving minimality. In order to obtain efficiency the functions maintain the signature mapping function as well (see Program 10.3.1). The first function implements the state cloning according Definition 10.4.2.

```
1  import ← "Program_10.3.1.cm";
2  clone : 𝒟ℱ𝒮𝒜×𝒮𝒯𝒜𝒯ℰ×𝒮𝒴ℳℬ𝒪ℒ×𝒮𝒯𝒜𝒯ℰ → 𝒟ℱ𝒮𝒜×𝒮𝒯𝒜𝒯ℰ;
3  clone((Σ, Q, q0, F, δ), p, a, q) := ((Σ, Q ∪ {q'}, q0, F', δ'), q'), where
4      q' := max(Q) + 1;
5      F' := F ∪ { {q'}   if q ∈ F
                 { ∅     otherwise ;
6      δ' := δ \ {((p, a), q)} ∪ {((p, a), q')} ∪ {((q', σ), δ(q, σ)) | σ ∈
   Σ & !δ(q, σ)};
7      ;
```

For a given minimal automaton $\mathcal{A}$ and a word α the function increaseExcept presented below returns the automaton $\mathcal{A}'$ that is minimal except for α or the longest prefix of α in $L(\mathcal{A})$ and is equivalent to $\mathcal{A}$.

Using an induction (Lines 12-29) we inductively proceed forward until we have treated α or are not able to continue in the automaton (Line 18). We start from the initial state and the given automaton (Line 14). After the k-th step, an intermediate automaton $\mathcal{A}'^{(k)}$ is obtained. We proceed by analyzing the state q' reached from the start state with the first $k+1$ symbols of α (Line 16). We distinguish two cases (Line 17):

Case 1. q' is not convergent in $\mathcal{A}'^{(k)}$. In this case we make no changes to the automaton (cf. Proposition 10.4.1).

Case 2. q' is convergent in $\mathcal{A}'^{(k)}$. In this case we obtain $\mathcal{A}'^{(k+1)}$ by cloning the state q' (cf. Proposition 10.4.4).

8 increaseExcept : $\mathcal{DFSA} \times \mathcal{WORD} \times (\mathcal{STSIGN} \to \mathcal{STATE}) \to \mathcal{DFSA} \times (\mathcal{STSIGN} \to \mathcal{STATE})$;
9 increaseExcept$((\Sigma, Q, q_0, F, \delta), \alpha, h) := (A', h')$, **where**
10 $\quad \delta_1 := \mathcal{F}_{2\to 1}(\delta)$;
11 $\quad q \in \mathcal{STATE}$;
12 $\quad (A', q, h') :=$ **induction**
13 $\quad\quad$ **step** 0 :
14 $\quad\quad\quad (A'^{(0)}, q^{(0)}, h'^{(0)}) := ((\Sigma, Q, q_0, F, \delta), q_0, h)$;
15 $\quad\quad$ **step** $k+1$:
16 $\quad\quad\quad q' := \mathrm{Proj}_5(A'^{(k)})(q^{(k)}, (\alpha)_{k+1})$;
17 $\quad\quad\quad ((A'^{(k+1)}, q^{(k+1)}), h'^{(k+1)}) :=$

$$\begin{cases} ((A'^{(k)}, q'), h'^{(k)} \setminus \{(\mathrm{calcSignature}(A'^{(k)}, q'), q')\}) & \textbf{if } |\delta_1(q')| = 1 \\ (\mathrm{clone}(A'^{(k)}, q^{(k)}, (\alpha)_{k+1}, q'), h'^{(k)}) & \textbf{otherwise} \end{cases};$$

18 $\quad\quad$ **until** $(k = |\alpha|) \vee \neg(!\,\mathrm{Proj}_5(A'^{(k)})(q^{(k)}, (\alpha)_{k+1}))$
19 $\quad\quad$;
20 $\quad$;

The next function addWord adds a word to the minimal automaton following Proposition 10.1.8 and the method described above.

21 addWord : $\mathcal{DFSA} \times (\mathcal{STSIGN} \to \mathcal{STATE}) \times \mathcal{WORD} \to \mathcal{DFSA} \times (\mathcal{STSIGN} \to \mathcal{STATE})$;
22 addWord$((\Sigma, Q, q_0, F, \delta), h, v) := (A'', h'')$, **where**
23 $\quad ((\Sigma', Q', q_0', F', \delta'), h') :=$ increaseExcept$((\Sigma, Q, q_0, F, \delta), v, h)$;
24 $\quad \tau :=$ stateseq(δ', q_0', v);
25 $\quad k := |\tau| - 1$;
26 $\quad T' := \langle \max(Q') + 1, \ldots, \max(Q') + |v| - k \rangle$;
27 $\quad \tau' := \tau \cdot T'$;
28 $\quad (A'', h'') :=$
reduceExcept$((\Sigma', Q' \cup \mathrm{set}(T'), q_0', F' \cup \{(\tau')_{|\tau'|}\}, \delta''), v, 0, h')$, **where**
29 $\quad\quad \delta'' := \delta' \cup \{(((\tau')_i, (v)_i), (\tau')_{i+1}) \mid i \in \langle k+1, \ldots, |v| \rangle\}$;
30 $\quad\quad$;
31 $\quad$;

The function deleteUnusedStates removes the unused states in accordance with Proposition 10.4.6. Starting from the last state on the path it tests its finality and the presence of outgoing transitions in Line 44. In the negative case it removes the state (Line 41) otherwise the induction terminates. Finally the function deleteWord deletes a word from from a minimal automaton performing the three steps described in the procedure above.

```
32  deleteUnusedStates : DFSA × WORD → DFSA × ℕ;
33  deleteUnusedStates((Σ, Q, q0, F, δ), v) := (A, m), where
34      τ := stateseq(δ, q0, v);
35      (A, m) := induction
36          step 0 :
37              (A^(0), m^(0)) := ((Σ, Q, q0, F \ {(τ)_{|v|+1}}, δ), |τ| − 1);
38          step i + 1 :
39              (Σ, Q', q0, F', δ') := A^(i);
40              q_i := (τ)_{|τ|−i−1};
41              v_i := (v)_{|τ|−i−1};
42              A^(i+1) := (Σ, Q' \ {q_i}, q0, F', δ' \ {((q_i, v_i), δ'(q_i, v_i))});
43              m^(i+1) := m^(i) − 1;
44          until ((τ)_{|τ|−i} ∈ F) ∨ ∃σ ∈ Σ : (! Proj_5(A^(i))((τ)_{|τ|−i}, σ))
45          ;
46      ;
47  deleteWord : DFSA × (STSIGN → STATE) × WORD →
    DFSA × (STSIGN → STATE);
48  deleteWord(A, h, v) := reduceExcept(A'', (v)_{1,...,m}, 0, h'), where
49      (A', h') := increaseExcept(A, v, h);
50      (A'', m) := deleteUnusedStates(A', v);
51      ;
```

Remark 10.4.9 The complexity of adding and deleting a word v by Program 10.4.8 is $O(|Q||v| \log |Q|)$. The factor $|Q|$ is due to the calculation of the reverse transition function δ' in Line 10. This calculation can be easily avoided by maintaining the number of incoming transitions for each state, which can be done in constant time. In this way the worst case time complexity can be reduced to $O(|v| \log |Q|)$.

10.5 The minimal subsequential transducer for a finite two-sided dictionary

Entries of lexica often come with additional information. This extended form of a lexicon can be formalized as a finite mapping $D : \Sigma^+ \to \Sigma^*$ between words, which represents a regular function. A mapping D of this form is also called a *two-sided dictionary*. In this section we show how to incrementally build the minimal subsequential transducer for a two-sided dictionary D.

Formal construction

The presentation in this section follows the scheme for computing the minimal deterministic automaton for a simple lexicon presented above by amending the outputs to the construction. We follow the approach in [Mihov and Maurel, 2001]. In order to understand the following construction it helps to recall that the minimization procedure for subsequential finite-state transducers in Section 5.4 was based on two steps: after converting the transducer to a specific canonical form (Definition 5.5.2) where output is produced on each path as early as possible, we applied a pseudo-minimization procedure. Throughout this section by a subsequential transducer we always mean a classical subsequential finite-state transducer with initial output as defined in Definition 5.4.4.

Convergent states of a subsequential transducer are defined as in the case of deterministic automata, cf. Definition 10.1.1. The following definitions and results generalize similar results used above for the case of one-sided dictionaries. New aspects are the output and the canonicity of the transducers.

Definition 10.5.1 Let $\mathcal{T} = \langle \Sigma, Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ be a trimmed subsequential finite-state transducer with initial output. $\mathcal{T}$ is *minimal except for* $w = a_1 a_2 \dots a_k \in \Sigma^*$ if

1. w is a prefix of a word in $L_{\times 2}(\mathcal{T})$. We consider the unique path in $\mathcal{T}$

   $$\tau_0 \rightarrow^{\langle a_1, \lambda_1 \rangle} \tau_1 \dots \rightarrow^{\langle a_k, \lambda_k \rangle} \tau_k, \text{ where } \tau_0 = q_0.$$

   with sequence of states

   $$T_w = \{\tau_0, \tau_1, \dots, \tau_k\}.$$

2. Distinct states in $Q \setminus T_w$ are always non-equivalent.

3. There are no convergent states in T_w.

4. The subsequential transducer $\mathcal{T}$ is in canonical form.

In what follows, $\mathcal{T} = \langle \Sigma, Q, q_0, F, \delta, \lambda, \iota, \Psi \rangle$ denotes a trimmed subsequential transducer. We refer to the notion of equivalent states in a subsequential transducer introduced in Definition 5.1.8.

Proposition 10.5.2 *Let $\mathcal{T}$ be minimal except for $w = a_1 a_2 \dots a_k \in \Sigma^+$, let $T_w = \{\tau_0, \tau_1, \dots, \tau_k\}$ denote the sequence of states for w. If no state in $Q \setminus T_w$ is equivalent to τ_k, then $\mathcal{T}$ is minimal except for $w' = a_1 a_2 \dots a_{k-1}$.*

Proof. In this case also distinct states in $Q \setminus \{\tau_0, \tau_1, \dots, \tau_{k-1}\}$ are always non-equivalent. The proposition directly follows from Definition 10.5.1. □

Proposition 10.5.3 *Let $\mathcal{T}$ be minimal except for $w = a_1 a_2 \dots a_k \in \Sigma^+$, let $T_w = \{\tau_0, \tau_1, \dots, \tau_k\}$ denote the sequence of states for w. Assume that τ_k is equivalent to a state $r \in Q \setminus T$. Then the subsequential transducer*

$$\mathcal{T}' = \left\langle \Sigma, Q \setminus \{\tau_k\}, q_0, F \setminus \{\tau_k\}, \delta', \lambda \setminus \lambda|_{\{\tau_k\} \times \Sigma}, \iota, \Psi \setminus \Psi|_{\{\tau_k\}} \right\rangle$$

where $\delta' = \delta|_{Q \setminus \{\tau_k\} \times \Sigma} \setminus \{\langle \tau_{k-1}, a_k, \tau_k \rangle\} \cup \{\langle \tau_{k-1}, a_k, r \rangle\}$ is minimal except for $w' = a_1 a_2 \dots a_{k-1}$ and $L(\mathcal{T}) = L(\mathcal{T}')$.

Proof. Redirecting a transition to an equivalent state does not change the output function of the transducer. Afterwards τ_k is not reachable, removing it again does not change the output function. Clearly, $\mathcal{T}'$ is trimmed. Since by definition equivalent states have the same input-output behaviour (cf. Definition 5.1.8) it is simple to see that $\mathcal{T}'$ is again in canonical form. Since the conditions in Definition 10.5.1 hold, the transducer $\mathcal{T}'$ is minimal except for w'. □

Proposition 10.5.4 *Let $\mathcal{T}$ be minimal except for $w = a_1a_2\dots a_k \in \Sigma^*$, let $T_w = \{\tau_0, \tau_1, \dots, \tau_k\}$ denote the sequence of states for w. Then τ_k is equivalent to $r \in Q \setminus T_w$ iff*

1. *$\tau_k \in F$ iff $r \in F$,*
2. *$\tau_k \in F$ implies that $\Psi(\tau_k) = \Psi(r)$,*
3. *$\forall \sigma \in \Sigma$: both $\delta(\tau_k, \sigma), \delta(r, \sigma)$ are undefined, or $\delta(\tau_k, \sigma) = \delta(r, \sigma)$ & $\lambda(\tau_k, \sigma) = \lambda(r, \sigma)$.*

Proof. The proposition follows from Proposition 5.5.8 and the fact that all transitions from τ_k and r lead to states in $Q \setminus T_w$ and there are no equivalent states in $Q \setminus T_w$. □

Proposition 10.5.5 *Let $\mathcal{T}$ be minimal except for ε and $\mathcal{T}$ be acyclic. Then $\mathcal{T}$ is a minimal subsequential transducer.*

Proof. Since the transducer is trimmed and acyclic there is no state equivalent to the initial state and therefore there are no equivalent states in the transducer. Hence the transducer $\mathcal{T}$ is minimal. □

In the next proposition we consider the situation where we extend the set of transitions departing from a node τ_k in order to add a new final word v to the transducer. In this situation, in order to preserve canonicity of the transducer we need to modify the outputs on the path leading to τ_k. In general the output of the new word v means that outputs that could be printed at early transitions before now need to be postponed since they are not compatible with the output for v. We have to recalculate the maximal possible output that is valid at each point of the path for all continuations. This leads to the updates λ', ι', Ψ' of the transition output function and the state output function in the proposition.

Proposition 10.5.6 *Let $\mathcal{T}$ be minimal except for $w = a_1a_2\dots a_k \in \Sigma^*$, let $T_w = \{\tau_0, \tau_1, \dots, \tau_k\}$ denote the sequence of states for w. Let*

$$v = a_1a_2\dots a_ka_{k+1}\dots a_m \in \Sigma^*, m > k$$

be a new word with output $\beta \in \Sigma^*$ such that $w = \max_{u \in L_{\times 2}(\mathcal{T})} u \wedge v$. Let $\Lambda_i = (\iota \cdot \lambda^*(q_0, a_1\dots a_i)) \wedge \beta$. Then

$$\mathcal{T}' := \langle \Sigma, Q \cup T', q_0, F \cup \{\tau_m\}, \delta', \lambda', \iota', \Psi' \rangle$$

where

- $T' = \{\tau_{k+1}, \ldots, \tau_m\}$ *is a set of new states* ($T' \cap Q = \emptyset$),
- $\delta' = \delta \cup \{\langle \tau_{i-1}, a_i, \tau_i \rangle \mid i = k+1, \ldots, m\}$,
- $\iota' = \iota \wedge \beta$,
- $$\begin{aligned} \lambda' \;=\; & \lambda|_{(Q\setminus T_w)\times\Sigma} \cup \{\langle \tau_{i-1}, a_i, \Lambda_{i-1}^{-1}\Lambda_i \rangle \mid i = 1, \ldots, k\} \\ \cup\; & \{\langle \tau_k, a_{k+1}, \Lambda_k^{-1}\beta \rangle\} \\ \cup\; & \{\langle \tau_{i-1}, a_i, \varepsilon \rangle \mid i = k+2, \ldots, m\} \\ \cup\; & \{\langle \tau_i, \sigma, \Lambda_i^{-1}(\iota \cdot \lambda^*(q_0, a_1 \ldots a_i\sigma)) \rangle \mid \sigma \in \Sigma, i = 0, \ldots, k \;\&\; \\ & \qquad\qquad \sigma \neq a_{i+1} \;\&\; !\lambda(\tau_i, \sigma)\}, \end{aligned}$$
- $\Psi' = \Psi|_{Q\setminus T_w} \cup \{\langle \tau_i, \Lambda_i^{-1} O_{\mathcal{T}}(a_1 \ldots a_i) \rangle \mid i = 1, \ldots, k \;\&\; \tau_i \in F\} \cup \{\langle \tau_m, \varepsilon \rangle\}$

is a subsequential transducer which is minimal except for v *and* $L(\mathcal{T}') = L(\mathcal{T}) \cup \{\langle v, \beta \rangle\}$.

Proof. Since w is the longest common prefix of v and a word from the input language of $\mathcal{T}$, the transition $\delta(\tau_k, a_{k+1})$ is not defined and therefore δ' and λ' are functional and $\mathcal{T}'$ is a subsequential transducer. Adding the state sequence T' - the last state τ_m being a final state - is adding v to the input language of the automaton. The definition of the initial output ι', the output function λ' and the state output Ψ' ensures that the transducer $\mathcal{T}'$ is in canonical form and that the output language remains the same up to the addition of output β for v (cf. Definitions 5.5.1, 5.5.2 and Proposition 5.5.6). Since the conditions in Definition 10.5.1 hold $\mathcal{T}'$ is a subsequential transducer minimal except for v and its language is $L(\mathcal{T}) \cup \{\langle v, \beta \rangle\}$. □

Adding the entries of the dictionary in alphabetic order. For a given two-sided dictionary $D = \{\langle \alpha^{(i)}, \beta^i \rangle \mid i = 1, \ldots, |D|\}$ such that $\alpha^{(i)}$ precedes lexicographically $\alpha^{(i+1)}$ for every $i < |D|$, the construction proceeds inductively in the following way:

1. For the first entry $\langle \alpha^{(1)}, \beta^{(1)} \rangle$. where $\alpha^{(1)} = a_1 a_2 \ldots a_k$ we construct the trivial subsequential transducer $\mathcal{T}^{(1)} = \langle \Sigma, T, \tau_0, \{\tau_k\}, \delta, \lambda, \beta^{(1)}, \{\langle \tau_k, \varepsilon \rangle\} \rangle$, where
   $$\begin{aligned} T &:= \{\tau_0, \tau_1, \ldots, \tau_k\} \text{ is a set of } k+1 \text{ new states,} \\ \delta &:= \{\langle \tau_{i-1}, a_i, \tau_i \rangle \mid i = 1, \ldots, k\}, \\ \lambda &:= \{\langle \tau_{i-1}, a_i, \varepsilon \rangle \mid i = 1, \ldots, k\}, \end{aligned}$$
   This trimmed subseqential transducer is minimal except $\alpha^{(1)}$ and represents the language $\{\langle \alpha^{(1)}, \beta^{(1)} \rangle\}$.
2. Let $\mathcal{T}^{(n)}$ be the transducer which is minimal except $\alpha^{(n)}$ and represents the language $\{\langle \alpha^{(i)}, \beta^{(i)} \rangle \mid i = 1, \ldots, n\}$.
3. As a pre-step for adding the next pair $\langle \alpha^{(n+1)}, \beta^{(n+1)} \rangle$ we treat those states τ_i of the state sequence $T_{\alpha^{(n)}}$ where $plab(\tau_i)$ is *not* a prefix of $\alpha^{(n+1)}$. In other words, we treat the states of $T_{\alpha^{(n)}}$ that do not belong to the future path $T_{\alpha^{(n+1)}}$. We apply Propositions 10.5.2 and 10.5.3 (depending on the (non-) existence of a state equivalent to the state to be treated) in an iterated manner until we obtain the transducer $\mathcal{T}' = \langle \Sigma, Q, q_0, F, \delta, \lambda, \Psi \rangle$

that is minimal except for $w = \alpha^{(n)} \wedge \alpha^{(n+1)}$. Equivalence checking is performed using Proposition 10.5.4.

4. From Proposition 10.1.9 we know that $w = \max_{u' \in L_{\times 2}(\mathcal{T}^{(n)})} u' \wedge \alpha^{(n+1)}$. Applying Proposition 10.5.6 we construct the transducer $\mathcal{T}^{(n+1)}$ that is minimal except for $\alpha^{(n+1)}$ and represents the language

$$\{\langle \alpha^{(i)}, \beta^{(i)} \rangle \mid i = 1, \ldots, n+1\}.$$

5. Until the set D is exhausted we increase n by one and loop through points 2-4.

6. Finally we apply Propositions 10.5.2 and 10.5.3 until we obtain the transducer $\mathcal{T}$ that is minimal except for ε and represents D. From Proposition 10.5.5 we know that $\mathcal{T}$ is minimal.

C(M) implementation for building the dictionary transducer

Program 10.5.7 The next program presents an efficient implementation for constructing the minimal subsequential transducer representing a finite two-sided dictionary given as input. The implementation proceeds similarly as Program 10.3.1.

The new program in Line 2 introduces state signatures for transducers. The signature of a state p is a triple. The first component says if p is a final state (Boolean value). The second component is the state output of the state in case it is a final state and the empty word otherwise. The third component characterizes the outgoing transitions of p and their corresponding outputs. Only transitions are taken into account that lead to states that are already "minimized" and will not be modified again by the algorithm. The function calcTSignature calculates the signature for a given state q in the transducer $(\Sigma, Q, q_0, F, \delta, \lambda, \iota, \Psi)$ (Lines 3-4).

```
1  import ← "Section_8.3.cm";
2  TSTSIGN is IB × WORD × 2^(SYMBOL×WORD×STATE);
3  calcTSignature : SSFSTI × STATE → TSTSIGN;
4  calcTSignature((Σ, Q, q0, F, δ, λ, ι, Ψ), q) :=
     (q ∈ F, if(q ∈ F, Ψ(q), ε), {(c, λ(q, c), δ(q, c)) | c ∈ Σ & !δ(q, c)});
```

The following function reduceTExcept proceeds in the same way as the function reduceExcept in Program 10.3.1. The only differences are the usage of the transducer state signatures and the updating of the output functions (Lines 23, 24).

```
5  reduceTExcept : SSFSTI × WORD × IN × (TSTSIGN → STATE) →
   SSFSTI × (TSTSIGN → STATE);
6  reduceTExcept(T, α, l, h) := (T', h'), where
7      τ := stateseq(δ, q0, α), where
8          (Σ, Q, q0, F, δ, λ, ι, Ψ) := T;
9          ;
```

```
10      (T', h') := induction
11          step 0 :
12              T'^(0) := T;
13              h'^(0) := h;
14          step i + 1 :
15              p := |τ| − i;
16              h_q := calcTSignature(T'^(i), (τ)_p);
17              (T'^(i+1), h'^(i+1)) :=
18                  case ¬(! h'^(i)(h_q)) : (T'^(i), h'^(i) ∪ {(h_q, (τ)_p)})
19                  otherwise :
    ((Σ', Q' \ {(τ)_p}, q'_0, F' \ {(τ)_p}, δ'', λ'', ι', Ψ''), h'^(i)), where
20                      (Σ', Q', q'_0, F', δ', λ', ι', Ψ') := T'^(i);
21                      r := h'^(i)(h_q);
22                      δ'' := (δ' \ {(((τ)_p, σ), δ'((τ)_p, σ)) | σ ∈ Σ' & ! δ'((τ)_p, σ)})\
                        {(((τ)_{p−1}, (α)_{p−1}), (τ)_p)} ∪ {(((τ)_{p−1}, (α)_{p−1}), r)};
23                      λ'' := λ' \ {(((τ)_p, σ), λ'((τ)_p, σ)) | σ ∈ Σ' & ! δ'((τ)_p, σ)};
24                      Ψ'' := Ψ' \ { {((τ)_p, Ψ'((τ)_p))}   if (τ)_p ∈ F'
                                    { ∅                      otherwise      ;
25                  ;
26          until |τ| − i = l + 1
27          ;
28      ;
```

The main function Tdic constructs the transducer T for a two-sided dictionary D given as a lexicographically ordered list of word pairs.[2]. Again as in the function mdic in Program 10.3.1 we start with the trivial transducer for the first pair of words (Lines 33-40).

```
29  Tdic : 𝒜ℒ𝒫ℋ𝒜ℬℰ𝒯 × (𝒲𝒪ℛ𝒟 × 𝒲𝒪ℛ𝒟)* → 𝒮𝒮ℱ𝒮𝒯ℐ;
30  Tdic(Σ_D, D) := Proj_1(reduceTExcept(T, Proj_1((D)_{|D|}), 0, h)), where
31      (T, h) ∈ 𝒮𝒮ℱ𝒮𝒯ℐ × (𝒯𝒮𝒯𝒮ℐ𝒢𝒩 → 𝒮𝒯𝒜𝒯ℰ);
32      (T, h) := induction
33          step 1 :
34              (α, β) := (D)_1;
35              τ := ⟨1, ..., |α| + 1⟩;
36              T^(1) := (Σ_D, set(τ), (τ)_1, {(τ)_{|α|+1}}, δ, λ, β, {((τ)_{|τ|}, ε)}), where
37                  δ := {(((τ)_i, (α)_i), (τ)_{i+1}) | i ∈ {1, ..., |α|}};
38                  λ := {(((τ)_i, (α)_i), ε) | i ∈ {1, ..., |α|}};
39                  ;
40              h^(1) := ∅;
```

Afterwards the induction in Lines 41-75 proceeds by adding the next pair of words to the transducer according to Proposition 10.5.6. Here the most complicated detail is the redefinition of the outputs on the path of the longest common prefix τ, which is needed to maintain canonicity. This is done in the subinduction in Lines 55-73. In this subinduction $c^{(i)}$ represents $\Lambda_{i-1}^{-1}\Lambda_i$, $l^{(i)}$ represents $\Lambda_i^{-1}(\iota \cdot \lambda^*(q_0, a_1 \dots a_i))$ and $b^{(i)}$ represents $\Lambda_i^{-1}\beta$ (cf. Proposition 10.5.6).

[2]The lexicographic order refers to the first components.

step $n+1$:
$(v, \beta) := (D)_{n+1}$;
$(u, \beta_u) := (D)_n$;
$\tau := \text{stateseq}(\delta, q_0, v)$, **where**
$(\Sigma, Q, q_0, F, \delta, \lambda, \iota, \Psi) := T^{(n)}$;
;
$k := |\tau| - 1$;
$((\Sigma', Q', q_0', F', \delta', \lambda', \iota', \Psi'), h^{(n+1)}) :=$
$\text{reduceTExcept}(T^{(n)}, u, k, h^{(n)})$;
$T' := \langle |Q'| + 1, \ldots, |Q'| + |v| - k \rangle$;
$\tau' := \tau \cdot T'$;
$T^{(n+1)} :=$
$(\Sigma', Q' \cup \text{set}(T'), q_0', F' \cup \{(T')_{|T'|}\}, \delta'', \lambda''', \iota'', \Psi'' \cup \{((T')_{|T'|}, \varepsilon)\})$, **where**
$\delta'' := \delta' \cup \{(((\tau')_i, (v)_i), (\tau')_{i+1}) \mid i \in \{k+1, \ldots, |v|\}\}$;
$\iota'' := \text{commonPrefix}(\iota', \beta)$;
$(\lambda'', \Psi'', c, l, b) \in$
$\mathcal{TRANSITIONOUT} \times \mathcal{STATEOUT} \times \mathcal{WORD} \times \mathcal{WORD} \times \mathcal{WORD}$;
$(\lambda'', \Psi'', c, l, b) :=$ **induction**
step 0 :
$(\lambda''^{(0)}, \Psi''^{(0)}, c^{(0)}, l^{(0)}, b^{(0)}) := (\lambda', \Psi', \varepsilon, \varepsilon, \beta)$;
step $i+1$:
$\lambda_i := \begin{cases} \iota' & \textbf{if } i = 0 \\ \lambda''^{(\mathrm{i})}((\tau)_i, (v)_i) & \textbf{otherwise} \end{cases}$;
$c^{(i+1)} := \text{commonPrefix}(l^{(i)} \cdot \lambda_i, b^{(i)})$;
$l^{(i+1)} := \text{remainderSuffix}(c^{(i+1)}, l^{(i)} \cdot \lambda_i)$;
$b^{(i+1)} := \text{remainderSuffix}(c^{(i+1)}, b^{(i)})$;
$X, L \in \mathcal{TRANSITIONOUT}$;
$X := \{(((\tau)_{i+1}, \sigma), \lambda''^{(\mathrm{i})}((\tau)_{i+1}, \sigma)) \mid \sigma \in \Sigma'$
$\& \ (\sigma \neq (v)_{i+1}) \wedge !\, \delta''((\tau)_{i+1}, \sigma)\}$;
$L := \lambda''^{(i)} \setminus X \cup \{((t, \sigma), l^{(i+1)} \cdot l_\sigma) \mid ((t, \sigma), l_\sigma) \in X\}$;
$\lambda''^{(i+1)} :=$
$\begin{cases} L & \textbf{if } i = 0 \\ L \setminus \{(((\tau)_i, (v)_i), \lambda_i)\} \cup \{(((\tau)_i, (v)_i), c^{(i+1)})\} & \textbf{otherwise} \end{cases}$;
$\Psi''^{(i+1)} :=$
case $(\tau)_{i+1} \in F'$:
$\Psi''^{(i)} \setminus \{((\tau)_{i+1}, \Psi_\tau)\} \cup \{((\tau)_{i+1}, l^{(i+1)} \cdot \Psi_\tau)\}$, **where**
$\Psi_\tau := \Psi''^{(\mathrm{i})}((\tau)_{i+1})$;
otherwise : $\Psi''^{(i)}$
;
until $i - 1 = k$
;
$\lambda''' :=$
$\lambda'' \cup \{(((\tau)_{k+1}, (v)_{k+1}), b)\} \cup \{(((\tau')_i, (v)_i), \varepsilon) \mid i \in \{k+2, \ldots, |v|\}\}$;
;
until $n = |D|$
;
;

Remark 10.5.8 The minimal subsequential transducer for a two-sided dictionary can be adapted by adding and deleting individual entries, applying a similar technique as the one presented in Section 10.4 [Skut, 2003].

Remark 10.5.9 The presented construction can be applied to dictionaries mapping words to numbers (cf. Section 5.6). The only difference is the use of minimum instead of largest common prefix and addition instead of concatenation for the target elements.

Remark 10.5.10 The indexing and search technology Lucene[3] uses the algorithm presented above for mapping terms to outputs[4].

[3] http://lucene.apache.org
[4] http://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html

Chapter 11

Constructing finite-state devices for text rewriting

A common programming task arising in many contexts is rewriting parts of a given input string to another form. Subparts of the input that match specific conditions are replaced by other output parts. In this way, the complete input string is translated to a new output form. Examples of this process include text tokenization, normalization or conversion of subexpressions, text highlighting, text annotation etc. Due to the relevance of text rewriting many programming languages offer matching and rewriting operations for subexpressions of strings. The way the rewrite rules are implemented in programming languages often goes beyond the range of regular languages (see [Friedl, 1997]). Still, regular expressions over $\Sigma^* \times \Sigma^*$, which are also called rewrite (or replace) rules, are powerful enough to model most interesting translation tasks. Using strictly regular relations and functions we can compose a cascade of replace rules into a single transducer. If the transducer is functional, then we can construct an equivalent bimachine or in some cases a subsequential transducer and thus achieve theoretically and practically optimal text processing speed. In this chapter we introduce basic constructions for building text rewriting transducers and bimachines from rewrite rules and provide implementations. First we present a simple realization of rewrite rules using regular relations following the approach in [Karttunen, 1997]. In general, this approach leads to an ambiguous form of text rewriting with several outputs. In the last section a more sophisticated construction for rewrite rules and relations is presented, which solves conflicts using the left-most-longest match strategy. We shall follow the methodology introduced in [Karttunen, 1996]. The application of regular relations for the realization of rewrite rules has been introduced by [Kaplan and Kay, 1994], it was further studied by [Mohri and Sproat, 1996, Mohri, 1997, Roche and Schabes, 1997a, Karttunen, 1997, Gerdemann and van Noord, 1999, Hulden, 2009].

This material has been / will be published by Cambridge University Press as *Finite State Techniques* by Stoyan Mihov and Klaus U. Schulz 2018. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works.
© Stoyan Mihov and Klaus U. Schulz 2018.

11.1 Simple text rewriting based on regular relations

In this section we look at binary regular string relations as introduced in Definition 4.5.2. For simplicity, the concatenation of string relations R, S is denoted $R.S$ (as opposed to "$R \dot{\cdot} S$" as in Definition 1.4.8). Recall that $Id(\Sigma^*)$ is a binary regular string relation (cf. Example 4.5.3).

Definition 11.1.1 Let a regular relation $T \subseteq \Sigma^* \times \Sigma^*$ be given. Then the *optional rewrite relation* $R^{opt}(T)$ *induced by* T is

$$R^{opt}(T) := Id(\Sigma^*).(T.Id(\Sigma^*))^*.$$

Since $R^{opt}(T)$ is constructed from T by applying identity (cf. Proposition 4.2.1), concatenation, and Kleene star on regular languages we obtain the following corollary.

Corollary 11.1.2 *For every regular relation* $T \subseteq \Sigma^* \times \Sigma^*$ *the optional rewrite relation* $R^{opt}(T)$ *induced by* T *is regular.*

Since in the above expression T is surrounded by expressions $Id(\Sigma^*)$, any substring of a text t in the domain of T can be replaced by its T-translations using $R^{opt}(T)$. The final asterisk in the definition of $R^{opt}(T)$ ensures that any number of substrings from $dom(T)$ can be translated. The second $Id(\Sigma^*)$ expression shows that subtexts between consecutive elements of $dom(T)$ are translated via identity. Text parts matching $Id(\Sigma^*)$ expressions can contain entries from $dom(T)$, as a consequence the T-replacement is always optional. The following proposition formalizes this property.

Proposition 11.1.3 *Let* $t \in \Sigma^*$ *be a text and* $T \subseteq \Sigma^* \times \Sigma^*$ *be a regular relation. Let* $R^{opt}(T)$ *be the optional rewrite relation induced by* T*. Let* $t, t' \in \Sigma^*$*. Then* $\langle t, t' \rangle \in R^{opt}(T)$ *iff there exist decompositions*

$$\begin{aligned} t &= u_0 v_1 u_1 v_2 u_2 \dots v_n u_n \\ t' &= u_0 v'_1 u_1 v'_2 u_2 \dots v'_n u_n \end{aligned}$$

such that $u_0, u_i \in \Sigma^*, v_i \in dom(T)$*, and* $\langle v_i, v'_i \rangle \in T$ *for* $i = 1, 2, \dots, n$*.*

Since we can decompose any given text $t \in \Sigma^*$ in accordance with the above proposition we obtain the following corollary.

Corollary 11.1.4 *For each regular relation* $T \subseteq \Sigma^* \times \Sigma^*$ *we always have* $dom(R^{opt}(T)) = \Sigma^*$*.*

Since a binary regular string relation T can offer an infinite number of translations for an input string (cf. Example 4.5.4) in general $R^{opt}(T)$ provides an infinite number of translations for a given text t.

When using the optional rewrite relation induced by a regular string relation T, strings from the domain of T need not to be replaced. In most applications it is desired that all entries of $dom(T)$ contained in a text t are replaced by their corresponding T-images. The following relation defines the obligatory rewriting induced by T.

Definition 11.1.5 Let $T \subseteq \Sigma^* \times \Sigma^*$ be a regular relation. Then the *obligatory rewrite relation* $R(T)$ *induced by* T is:

$$R(T) := N(T).(T.N(T))^*,$$

where $N(T)$ is the identity function on all strings that do not contain any infix from $dom(T)$, including $\{\langle \varepsilon, \varepsilon \rangle\}$:

$$N(T) = Id(\Sigma^* \setminus (\Sigma^*.dom(T).\Sigma^*)) \cup \{\langle \varepsilon, \varepsilon \rangle\}.$$

Again, since $R(T)$ is constructed from T and $\langle \varepsilon, \varepsilon \rangle$ by applying identity, projection, union, concatenation, Kleene star on regular languages, and set difference on regular languages over Σ^*, we obtain the following corollary (cf. Point 2 in Example 4.5.3).

Corollary 11.1.6 *For every regular relation* $T \subseteq \Sigma^* \times \Sigma^*$ *the obligatory rewrite relation* $R(T)$ *induced by* T *is regular.*

In the definition of $R(T)$ only substrings that do not contain entries from $dom(T)$ are translated via identity. Entries of $dom(T)$ are always translated. Yet it should be noted that entries of $dom(T)$ in a text t can overlap, hence in general there is not a unique decomposition of a given text into substrings from $dom(T)$ on the one-hand side and substrings that do not contain substrings in $dom(T)$ on the other hand. The following proposition formalizes these aspects.

Proposition 11.1.7 *Let* $t \in \Sigma^*$ *be a text and* $T \subseteq \Sigma^* \times \Sigma^*$ *be a regular relation. Let* $R(T)$ *be the obligatory rewrite relation induced by* T*, let* $N(T)$ *be defined as above. Let* $t, t' \in \Sigma^*$*. Then* $\langle t, t' \rangle \in R(T)$ *iff there exist decompositions*

$$\begin{aligned} t &= u_0 v_1 u_1 v_2 u_2 \dots v_n u_n \\ t' &= u_0 v'_1 u_1 v'_2 u_2 \dots v'_n u_n \end{aligned}$$

such that $u_0, u_i \in dom(N(T))$*,* $v_i \in dom(T)$*, and* $\langle v_i, v'_i \rangle \in T$ *for* $i = 1, 2, \dots, n$*.*

Again we can decompose any given text $t \in \Sigma^*$ in the way described in the proposition and obtain the following corollary.

Corollary 11.1.8 *For every regular relation* $T \subseteq \Sigma^* \times \Sigma^*$ *we always have* $dom(R(T)) = \Sigma^*$*.*

Example 11.1.9 Let $\Sigma = \{a, b, c, d\}$ and $T = \{\langle ab, d \rangle, \langle bc, d \rangle\}$. Then the optional and obligatory rewrite relations induced by T are shown in Figure 11.1.

11.2 Using deterministic machines for simple text rewriting

We now ask under which circumstances subsequential transducers and bimachines can be used for the above form of text rewriting (or restricted versions), given a regular relation T as above. Obviously, with such deterministic devices,

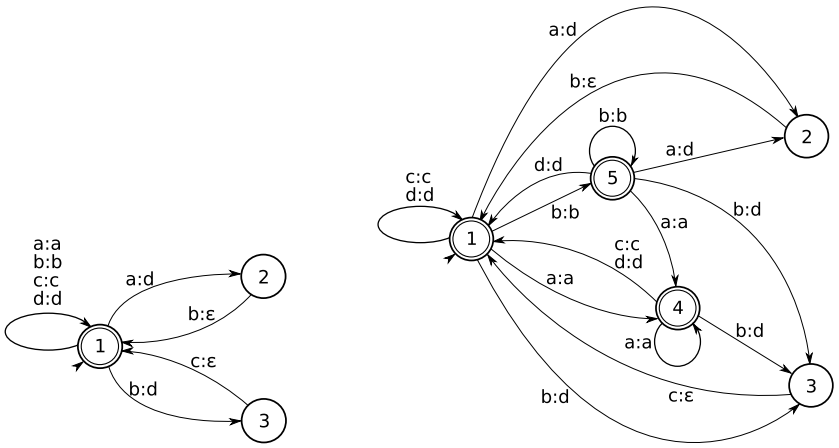

Figure 11.1: The transducers representing the optional (lhs) and obligatory (rhs) rewrite relations for the relation $T = \{\langle ab, d\rangle, \langle bc, d\rangle\}$.

only text rewriting *functions* can be built. As a matter of fact the optional rewrite relation $R^{opt}(T)$ is not functional unless T is the identity function (or its restriction). Hence, using deterministic devices is mainly interesting for obligatory text rewriting and the relation $R(T)$. The functionality of a rewrite relation $R(T)$ can be effectively tested using the method presented in Section 4.6 and implemented in Program 8.2.12. If $R(T)$ is functional, then a bimachine for the obligatory text rewriting relation $R(T)$ can be built using the construction presented in Section 6.2 and implemented in Program 8.4.1.

It is natural to ask if functionality of an obligatory text rewriting relation $R(T)$ can be read off directly from the base relation T. If T is not functional, then $R(T)$ is also not functional. Due to the fact that substrings of a text in $dom(T)$ can overlap in general the converse is not true. For example, the relation T in Example 11.1.9 is functional, but the relation $R(T)$ is not functional – we have $\langle abc, dc\rangle \in R(T)$ and $\langle abc, ad\rangle \in R(T)$. We now introduce a criterion for T that guarantees functionality of obligatory text rewriting with $R(T)$.

Definition 11.2.1 Let $\alpha, \beta \in \Sigma^*$ be two words. We say that α *overlaps* with β if there exists a nonempty word $\gamma \in \Sigma^+$ such that γ is a proper suffix of α and a proper prefix of β. We say that α *contains* β if $|\alpha| > |\beta|$ and there exist two words $\alpha', \alpha'' \in \Sigma^*$, such that $\alpha = \alpha'\beta\alpha''$. We say that the relation $T \subseteq \Sigma^* \times \Sigma^*$ is *conflict free* if for any two words $\alpha, \beta \in dom(T)$ it never happens that α overlaps with β or α contains β.

Proposition 11.2.2 *Let $T \subseteq \Sigma^* \times \Sigma^*$ be a regular relation such that T is functional and $\varepsilon \notin dom(T)$. If T is conflict free, then $R(T)$ is functional.*

Proof. Let us assume for the text $t \in \Sigma^*$ there exist $t', t'' \in \Sigma^*$ such that $\langle t, t'\rangle \in R(T)$, $\langle t, t''\rangle \in R(T)$ and $t \neq t'$. Since T is functional according to

Proposition 11.1.7 there exist two distinct decompositions of t

$$\begin{aligned} t &= u_0^{(1)}v_1^{(1)}u_1^{(1)}v_2^{(1)}u_2^{(1)}\dots v_n^{(1)}u_n^{(1)} \\ t &= u_0^{(2)}v_1^{(2)}u_1^{(2)}v_2^{(2)}u_2^{(2)}\dots v_n^{(2)}u_n^{(2)} \end{aligned}$$

such that $u_0^{(1)}, u_i^{(1)}, u_0^{(2)}, u_i^{(2)} \in dom(N(T))$ and $v_i^{(1)}, v_i^{(2)} \in dom(T)$ for $i = 1, 2, \dots, n$. Let $j \in \{0, \dots n\}$ be such that $u_0^{(1)}v_1^{(1)}u_1^{(1)}\dots v_j^{(1)} = u_0^{(2)}v_1^{(2)}u_1^{(2)}\dots v_j^{(2)}$ and $u_j^{(1)} \neq u_j^{(2)}$ or $v_{j+1}^{(1)} \neq v_{j+1}^{(2)}$.
Case 1: $u_j^{(1)} \neq u_j^{(2)}$. Without loss of generality we assume that $u_j^{(1)} < u_j^{(2)}$. The assumption $u_j^{(1)}v_{j+1}^{(1)} \leq u_j^{(2)}$ imples that $v_{j+1}^{(1)}$ is a substring of $u_j^{(2)}$, which contradicts $u_j^{(2)} \in dom(N(T))$. The assumption $u_j^{(1)}v_{j+1}^{(1)} > u_j^{(2)}$ implies that $v_{j+1}^{(1)}$ overlaps with $v_{j+1}^{(2)}$, which contradicts the conflict freeness of T.
Case 2: $u_j^{(1)} = u_j^{(2)}$ and $v_{j+1}^{(1)} \neq v_{j+1}^{(2)}$. Then either $v_{j+1}^{(1)}$ contains $v_{j+1}^{(2)}$ or $v_{j+1}^{(2)}$ contains $v_{j+1}^{(1)}$, which yields a contradiction to the conflict freeness of T. □

Below a C(M)-program will be given that tests conflict-freeness for base relations T that are finite sets of pairs of strings.

Realizing subfunctions for non-functional obligatory rewrite relations. Even if $R(T)$ is not functional we might be interested to build a bimachine that has the same domain as $R(T)$ (i.e., domain Σ^*) and represents a subset of $R(T)$. In Proposition 6.2.9 we have seen that such a bimachine can be built, provided that a *real-time* transducer representing $R(T)$ can be constructed. Proposition 4.4.8 shows that to this end we must assume that $R(T)$ is not infinitely ambigious. The question arises how we can recognize if $R(T)$ satisfies this assumption. If the base relation T contains an entry of the form $\langle \varepsilon, \alpha \rangle$ for $\alpha \neq \varepsilon$, then the obligatory rewrite relation $R(T)$ is infinitely ambiguous. At any position of an input text we can add an arbitrary number of substrings α. We now show that if T is not infinitely ambiguous and the empty string cannot be translated to a non-empty string, then $R(T)$ is not infinitely ambiguous.

Proposition 11.2.3 *Let $T \subseteq \Sigma^* \times \Sigma^*$ be a regular relation that is not infinitely ambiguous and assume that $\varepsilon \notin dom(T)$. Then $R(T)$ is not infinitely ambiguous and a bimachine can be built that translates every text $t \in \Sigma^*$ into an output string t' such that $\langle t, t' \rangle \in R(T)$.*

Proof. According to Proposition 11.1.7 for every $\langle t, t' \rangle \in R(T)$ there exist decompositions

$$\begin{aligned} t &= u_0v_1u_1v_2u_2\dots v_nu_n \\ t' &= u_0v_1'u_1v_2'u_2\dots v_n'u_n \end{aligned}$$

such that $u_0, u_i \in dom(N(T))$, $v_i \in dom(T)$, and $\langle v_i, v_i' \rangle \in T$ for $i = 1, 2, \dots, n$. Since n, $|u_i|$ and $|v_i|$ are bounded by $|t|$ there exist only a finite number of possible decompositions of t of the above form such that $u_0, u_i \in dom(N(T))$, $v_i \in dom(T)$. Since T is not infinitely ambiguous for each decomposition of t there exist only a finite number of corresponding decompositions of t' such that $\langle v_i, v_i' \rangle \in T$, Hence for every text t the number of rewritings t' is finite and therefore $R(T)$ is not infinitely ambiguous.

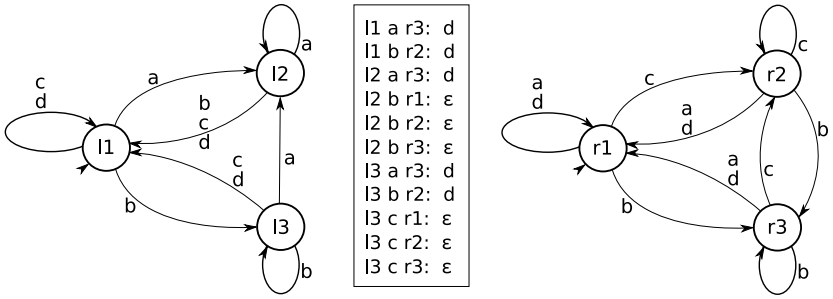

Figure 11.2: The bimachine constructed for the base relation $T = \{\langle ab, d\rangle, \langle bc, d\rangle\}$. The obligatory rewrite relation $R(T)$ is not functional. In the output function tuples where the input symbol is just copied are omitted.

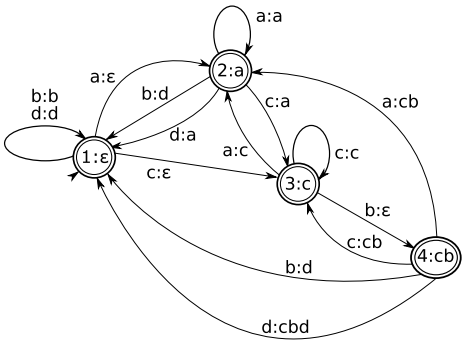

Figure 11.3: The minimal subsequential transducer constructed for obligatory text rewriting with base relation $T = \{\langle ab, d\rangle, \langle cbb, d\rangle, \langle ad, a\rangle\}$.

From Proposition 4.4.8 it follows that there exists a real-time transducer equivalent to $R(T)$ and therefore according to Proposition 6.2.9 we can build a bimachine $\mathcal{B}$ with domain Σ^* such that for every $t \in \Sigma^*$ we have $\langle t, O_{\mathcal{B}}(t)\rangle \in R(T)$. □

Figure 11.2 presents the bimachine constructed by Program 8.4.1 and pseudo-minimized by Program 8.4.2 for the non-functional rewrite rule of Example 11.1.9.

Using subsequential transducers for functional obligatory text rewriting. Let us come back to the situation where the obligatory relation $R(T)$ is functional. In order to find a subsequential transducer for $R(T)$, the relation $R(T)$ needs to have the bounded variation property (cf. Section 5.2). The bounded variation property of a functional rewrite relation $R(T)$ can be effectively tested using the method presented in Section 5.3 and implemented in Program 8.3.2. If $R(T)$ has the bounded variation property, then an equivalent subsequential transducer can be built using the construction presented in Section 5.2 and implemented in Program 8.3.1.

Note that the obligatory rewrite relation $R(T)$ induced by a finite conflict free base relation T always has the bounded variation property and thus a subse-

quential transducer representing $R(T)$ can be constructed. Figure 11.3 presents the minimal subsequential transducer constructed by Program 8.3.1 and minimized by Program 8.3.7 for the rewrite rule induced by the finite conflict free function

$$T = \{\langle ab, d\rangle, \langle cbb, d\rangle, \langle ad, a\rangle\}.$$

In practice, base relations T used for text rewriting tasks are often not conflict free. The construction of rewrite functions that perform a conflict resolution strategy to resolve ambiguities due to overlapping elements of $dom(T)$ will be the subject of the next section.

Program 11.2.4 Our next program presents the implementation of the construction of optional and obligatory rewrite relations from a source transducer according to Definitions 11.1.1, 11.1.5 and the decision procedure for testing conflict freeness and the functionality of the obligatory rewrite relation induced by a finite-state transducer T in accordance with Proposition 11.2.2.

```
1  import ← "Section_8.2.cm";
2  replaceOpt_FST, replace_FST : ALPHABET_T × FST_T → FST_T;
3  replaceOpt_FST(Σ, T) := concat_FST(N, star_FST(concat_FST(T, N))), where
4      N := identity_FSA(all(Σ));
5      ;
6  replace_FST(Σ, T) := concat_FST(NT, star_FST(concat_FST(T, NT))), where
7      NT := union_FST(identity_FSA(diff_FSA(all(Σ),
           concat_FSA(⟨all(Σ), domain_FST(T), all(Σ)⟩)))), FST_2WORD("", ""));
8      ;
9  testConflictFreeness : FST_T → IB;
10 testConflictFreeness(T) :=
       (|Proj_2(overlaps)| = 0) ∧ (|Proj_2(containments)| = 0), where
11     Σ := Proj_1(T);
12     dom_T := domain_FST(T);
13     plus_Σ := plus_FSA(symbolSet2FSA(Σ));
14     del_Σ := product_FSA(plus_Σ, FSA_WORD(""));
15     prefixes := range_FST(compose_FST(identity_FSA(dom_T),
                       concat_FST(identity_FSA(plus_Σ), del_Σ)));
16     suffixes := range_FST(compose_FST(identity_FSA(dom_T),
                       concat_FST(del_Σ, identity_FSA(plus_Σ))));
17     overlaps := trim_FSA(intersect_FSA(prefixes, suffixes));
18     leftCont := concat_FSA(⟨plus_Σ, dom_T, all(Σ)⟩);
19     rightCont := concat_FSA(⟨all(Σ), dom_T, plus_Σ⟩);
20     containments :=
   trim_FSA(intersect_FSA(dom_T, union_FSA(leftCont, rightCont)));
21     ;
22 testReplaceFunctionality : FST_T → IB;
23 testReplaceFunctionality(T) := testFunctionality_FST(T) ∧
   (|Proj_2(includesEpsilon)| = 0) ∧ testConflictFreeness(T), where
24     includesEpsilon := trim_FSA(intersect_FSA(FSA_WORD(""), domain_FST(T)));
25     ;
```

In Line 2 the types of the rewrite relation construction functions are defined.

The optional rewrite relation is defined in Lines 3-5 following Definition 11.1.1, in Lines 6-8 the obligatory rewrite relation is defined following Definition 11.1.5. Then in Lines 9-21 the test for conflict freeness is defined following Definition 11.2.1. In Line 11 Σ is set to the alphabet of the transducer T. Line 12 constructs the domain of T. Line 13 defines the automaton representing all non-empty words over Σ. In Line 14 we define the function that maps any non-empty word over Σ to the empty word ε. Lines 15 and 16 construct the automata representing the set of all non-empty proper prefixes respectively suffixes for the domain of T. In Line 17 the trimmed automaton representing the non-empty words that are simultaneously proper suffixes and prefixes is constructed. Lines 18 and 19 build the automata representing the set of words containing a word from the domain of T and having a nonempty word to the left or to the right, respectively. In Line 20 the trimmed automaton for all containments is constructed. According to Definition 11.2.1 T is conflict free if the domain of T has neither overlaps nor containments. This is checked in Line 10 by testing the number of states of the automata *overlaps* and *containments*. In Lines 22-25 the test for functionality of the obligatory rewrite relation induced by the transducer T is given. In line 24 we construct the trimmed automaton for the intersection of the empty word and the domain of T. According to Proposition 11.2.2 $R(T)$ is functional if T is functional, the domain of T does not include the empty word, and T is conflict free. This is checked in Line 23.

11.3 Left-most-longest-match text rewriting

In text rewriting, pieces of text we would like to replace ("originals") may overlap. Therefore we need a strategy to resolve this form of conflict. We now give a more formal description of the leftmost-longest match strategy for text rewriting. Under this strategy there is a preference for replacing originals that start first in the natural (left-to-right) reading order of the text. Among distinct originals with the same starting point, the longest one is replaced. In most cases, long originals carry more specific information than shorter units of text, which explains the intuitive appeal of this decision. If a part is replaced, other overlapping originals are excluded and search continues after the replaced part.

In Definition 1.3.7 we introduced the notion of an infix occurrence of a text t. Clearly, for a given text $t \in \Sigma^*$ there exists only a finite number of infix occurrences. We now give a definition for overlapping occurrences (cf. the definition of overlapping words given in Definition 11.2.1). Let us remind that for words $u, v \in \Sigma^*$ the notation $u \leq v$ ($u < v$) expresses that u is a (proper) prefix of v.

Definition 11.3.1 Two infix occurrences $\langle u_1, v_1, w_1 \rangle, \langle u_2, v_2, w_2 \rangle$ of the text t *overlap* if $u_1 < u_2 < u_1 v_1$. In this case we write $\langle u_1, v_1, w_1 \rangle <_{ov} \langle u_2, v_2, w_2 \rangle$. A set A of infix occurrences of the text t is said to be *non-overlapping* if two distinct infix occurrences of A never overlap.

Definition 11.3.2 Let A be a set of non-overlapping infix occurrences of the text $t \in \Sigma^*$, let $k = |A|$. We may represent A in the form

$$\{\langle u_1, v_1, w_1 \rangle, \ldots, \langle u_k, v_k, w_k \rangle\}$$

with indices $1, \ldots, k$ such that $u_j v_j < u_{j+1}$ for $1 \leq i < j < k$. The *canonical representation* of t for A is the sequence of $2k+1$ words $\langle u_1, v_1, x_2, v_2, \ldots, x_k, v_k, w_k \rangle$, where $x_i = [u_{i-1}v_{i-1}]^{-1}u_i$ for $2 \leq i \leq k$.[1]

Clearly, if $\langle u_1, v_1, x_2, v_2, \ldots, x_k, v_k, w_k \rangle$ is the canonical representation of a text t for a set A of non-overlapping infix occurrences, then $t = u_1 v_1 x_2 v_2 \cdots x_k v_k w_k$ and from the canonical representation we can reconstruct the set of infix occurrences A.

Example 11.3.3 Let $t = aaaaa$, let $A = \{\langle a, a, aaa \rangle, \langle aa, a, aa \rangle, \langle aaaa, a, \varepsilon \rangle\}$. The canonical representation of t for A is the sequence

$$\langle u_1, v_1, x_2, v_2, x_3, v_3, w_3 \rangle = \langle a, a, \varepsilon, a, aa, a, \varepsilon \rangle.$$

Definition 11.3.4 Let A, B be two sets of infix occurrences of the text t. We define the following functions:

$$\begin{aligned} &AFTER(A, B) = \{\langle u, v, w \rangle \in A \mid \forall \langle u', v', w' \rangle \in B : u'v' \leq u \wedge u' < u\} \\ &LEFTMOST(A) = \{\langle u, v, w \rangle \in A \mid \forall \langle u', v', w' \rangle \in A : u \leq u'\} \\ &LONGEST(A) = \{\langle u, v, w \rangle \in A \mid \forall \langle u', v', w' \rangle \in A : u \neq u' \vee v' \leq v\}. \end{aligned}$$

Definition 11.3.5 Let A be a set of infix occurrences of the text t. The *subset of leftmost-longest infix occurrences of* A is the set $LML(A) := \bigcup_{i=0}^{\infty} \mathcal{V}_i$ where

$$\begin{aligned} &\mathcal{V}_0 = \emptyset \\ &\mathcal{V}_{i+1} = \mathcal{V}_i \cup LONGEST(LEFTMOST(AFTER(A, \mathcal{V}_i))). \end{aligned}$$

Clearly $LML(A)$ is always finite and the chain $\mathcal{V}_0 \subseteq \mathcal{V}_1 \subseteq \mathcal{V}_2 \subseteq \ldots$ becomes constant after a finite number of steps. We always have $|\mathcal{V}_{i+1} \setminus \mathcal{V}_i| \leq 1$. Intuitively, applying $AFTER$ selects from all infix occurrences in A those where the central component starts after all central components of the set $\mathcal{V}_i$. Applying the $LEFTMOST$-operation then selects the occurrences where the central component has the leftmost position. For all occurrences left, the central component starts at the same position. Now $LONGEST$ selects the one with the longest central component.

Proposition 11.3.6 *Let A be set of infix occurrences of the text t. Then all infix occurrences in $LML(A)$ are non-overlapping.*

Definition 11.3.7 Let $T \subseteq \Sigma^* \times \Sigma^*$ be a relation and $t \in \Sigma^*$ be a text. Let $A_{dom(T)}$ be the set of all occurrences of words from $dom(T)$ in the text t. Let

$$\langle u_1, v_1, x_2, v_2, \ldots x_k, v_k, w_k \rangle$$

be the canonical representation of t for the set $LML(A_{dom(T)})$. Then the text t' is a *rewriting of t with T under the leftmost-longest match strategy* if

$$t' = u_1 v_1' x_2 v_2' \cdots x_k v_k' w_k$$

and $\langle v_i, v_i' \rangle \in T$ for $1 \leq i \leq k$. The *rewrite relation under the leftmost-longest match strategy* for T is the relation $R^{LML}(T)$ consisting of all pairs $\langle t, t' \rangle \in \Sigma^* \times \Sigma^*$ such that t' is a rewriting of t with T under the leftmost-longest match strategy.

[1] For the notation $^{-1}$, see Definition 1.3.4.

Corollary 11.3.8 *Let $T : \Sigma^+ \to \Sigma^*$ be a function. Then the rewrite relation $R^{LML}(T)$ under the leftmost-longest match strategy for T is functional.*

Remark 11.3.9 *Other strategies for resolving overlapping conflicts.* It is obvious that using variants of the above notions we could define in a similar way other characteristic subsets of non-overlapping occurrences for a given set of infix occurrences of a text (rightmost-longest, leftmost shortest, ...). Each such notion gives rise to its own way of resolving overlapping conflicts and to a special form of text rewriting. Though the algorithm presented below could be generalized, we only concentrate on the leftmost-longest match strategy.

11.4 Regular relations for left-most-longest match rewriting

The previous section introduced text rewriting under the left-most-longest match strategy in a general context. We now come back to finite-state technology and regular relations/functions for left-most-longest match rewriting. We consider the situation where the local replacement of "originals" is described in terms of a transducer. The following construction shows that in this situation also text rewriting under the left-most-longest match rewriting is again a regular process. Let us remark that we do not assume that the input transducer is functional. Text rewriting – even under the left-most-longest match strategy – in general defines a proper string *relation* since originals can be replaced by distinct strings. Clearly, for a *functional input* transducer T also text rewriting under the left-most-longest match strategy derived from T is functional.

Program 11.4.1 The following program introduces a function $\text{replaceLML}_{\text{FST}}(\Sigma, T)$ that takes as input a finite-state transducer T such that $\varepsilon \notin L_{\times 2}(T)$ and computes as output a transducer representing the relation $R^{LML}(R_T)$. The program thus describes a general method for realizing regular text rewriting under the leftmost-longest match strategy for a relation R_T given by a transducer T. The algorithm is based on [Karttunen, 1996].

In order to understand the following construction it makes sense to imagine text rewriting with T under the leftmost-longest match strategy as the result of applying in consecutive order four steps.

1. As a first step ("initial match"), using a new special symbol † (dagger), we mark in the input text all positions that represent the starting point of an infix in the domain of T. At this point it is not essential if the infix represents a leftmost, or a leftmost-longest infix occurrence. After introducing the markers, the infixes in the domain of T that follow a symbol † may be corrupted in the sense that they also have additional symbols †. This step corresponds to marking the beginning of the matches of all occurrences in the set A determined by the domain of T in Definition 11.3.5.

2. As a second step ("left-to-right"), the marked text obtained from Step 1 is transformed to a new form where both the beginning and the end of rewrite occurrences are marked by symbols $\ll$ and $\gg$, respectively. Between a

pair of corresponding markers $\ll$ and $\gg$, no other markers in $\{\dagger, \ll, \gg\}$ may appear. Before the first matched region, between two consecutive matched regions, and after the last matched region, only symbols from the original alphabet Σ may occur, ensuring in this way that no other possible matches exist. This step leads to several alternative output texts, each representing a leftmost match decomposition. One of these output texts exactly marks the regions relevant for the leftmost-*longest* match strategy. This step corresponds to the composition of the operations *AFTER* and *LEFTMOST* in Definition 11.3.5.

3. The third step ("longest match") selects the leftmost-longest decomposition. This step corresponds to the operation *LONGEST* in Definition 11.3.5. Note that for steps 1-3 only the domain of T is relevant.

4. The final step ("replacement") replaces marked regions by their T-image(s). All auxiliary markers are erased.

In the program below, each step is represented by a transducer. By composing the four transducers, text rewriting is achieved using a single transducer. We now present the program and fill out details.

```
1  import ← "Program_11.2.4.cm";
2  replaceLML_FST : ALPHABET* × FST → FST;
3  replaceLML_FST(Σ, T) :=
   compose_FST(⟨initialMatch, leftToRight, longestMatch, replacement⟩),
                where
```

The program imports Program 11.2.4 and defines the type of the main function in Lines 1-2. The main transducer is defined as the composition of the four transducers initialMatch, leftToRight, longestMatch and replacement, which will be defined below.

```
4      lb :='≪';
5      rb :='≫';
6      cb :='†';
7      assert ← Σ ∩ {lb, rb, cb} = ∅;
8      Σ_x := Σ ∪ {lb, rb, cb};
9      ALL := all(Σ_x);
10     not, contain : FSA → FSA;
11     not(L) := diff_FSA(ALL, L);
12     contain(L) := concat_FSA(⟨ALL, L, ALL⟩);
```

For preparing Step 1 ("left-to-right") we start in Lines 4-6 by defining the auxiliary marker symbols. As described above, the dagger symbol (†) is used for marking the starting positions of potential rewrite occurrences. The special left bracket ($\ll$) and the special right bracket ($\gg$) are used for enclosing regions of rewrite occurrences. These symbols only appear during the construction of the intermediate steps. In Line 7 we ensure that the auxiliary symbols do not appear in the target alphabet. The extended alphabet Σ_x where the auxiliary symbols are added is defined in Line 8. The automaton *ALL* recognized all

words over the extended alphabet (Line 9). In Lines 10-12 we define the functions not(L) and contain(L), which construct the complement of an automaton and the automaton recognizing all words that contain an occurrence of a word from the input automaton.

13 $\quad \text{intro}, \text{introx}, \text{xintro} : \mathcal{ALPHABET} \rightarrow \mathcal{FST};$
14 $\quad \text{intro}(S) := \text{star}_{\text{FST}}(\text{union}_{\text{FST}}(\text{identity}_{\text{FSA}}(\text{symbolSet2FSA}(\Sigma_x \setminus S)),$
$\qquad \text{product}_{\text{FSA}}(\text{FSA}_{\text{WORD}}(\text{""}), \text{symbolSet2FSA}(S))));$
15 $\quad \text{introx}(S) :=$
$\text{option}_{\text{FST}}(\text{concat}_{\text{FST}}(\text{intro}(S), \text{identity}_{\text{FSA}}(\text{symbolSet2FSA}(\Sigma_x \setminus S))));$
16 $\quad \text{xintro}(S) :=$
$\text{option}_{\text{FST}}(\text{concat}_{\text{FST}}(\text{identity}_{\text{FSA}}(\text{symbolSet2FSA}(\Sigma_x \setminus S)), \text{intro}(S)));$

As in [Kaplan and Kay, 1994] we define in Line 14 the function intro(S) that freely introduces symbols from the set S into an input string not containing symbols in S. The variants introx(S) and xintro(S) (Lines 15,16) introduce the symbols from S at any position except the end and the beginning of a string, respectively.

17 $\quad \text{ign}, \text{ignx}, \text{xign} : \mathcal{FSA} \times \mathcal{ALPHABET} \rightarrow \mathcal{FSA};$
18 $\quad \text{ign}(L, S) := \text{range}_{\text{FST}}(\text{compose}_{\text{FST}}(\text{identity}_{\text{FSA}}(L), \text{intro}(S)));$
19 $\quad \text{ignx}(L, S) := \text{range}_{\text{FST}}(\text{compose}_{\text{FST}}(\text{identity}_{\text{FSA}}(L), \text{introx}(S)));$
20 $\quad \text{xign}(L, S) := \text{range}_{\text{FST}}(\text{compose}_{\text{FST}}(\text{identity}_{\text{FSA}}(L), \text{xintro}(S)));$

Next, we define a family of ignore operators (Lines 17-20). For a given automaton L and a set of symbols S they construct the automaton which recognizes all words from L with freely introduced symbols from S. The variants ignx and xign do not permit introduction at the end (Line 19) and respectively at the beginning (Line 20).

21 $\quad \text{ifPthenS}, \text{ifSthenP}, \text{PiffS}, \text{LiffR} : \mathcal{FSA} \times \mathcal{FSA} \rightarrow \mathcal{FSA};$
22 $\quad \text{ifPthenS}(P, S) := \text{not}(\text{concat}_{\text{FSA}}(P, \text{not}(S)));$
23 $\quad \text{ifSthenP}(P, S) := \text{not}(\text{concat}_{\text{FSA}}(\text{not}(P), S));$
24 $\quad \text{PiffS}(P, S) := \text{intersect}_{\text{FSA}}(\text{ifPthenS}(P, S), \text{ifSthenP}(P, S));$
25 $\quad \text{LiffR}(L, R) := \text{PiffS}(\text{concat}_{\text{FSA}}(ALL, L), \text{concat}_{\text{FSA}}(R, ALL));$

Following [Kaplan and Kay, 1994] we define several filter automata to ensure that the markers introduced are placed in the correct positions. The filter ifPthenS(P, S) (Line 22) represents all words w where each prefix of w representing a string in P is followed by a suffix which is in S. Similarly, the filter ifSthenP(P, S) (Line 23) represents all words for which each suffix from S is preceeded by a prefix from P. The filter PiffS(P, S) (Line 24) combines the two given above. The filter LiffR(L, R) (Line 25) describes the words where every position is preceded by a string with a suffix in L iff it is followed by a string with a prefix in R.

26 $\quad D := \text{domain}_{\text{FST}}(T);$
27 $\quad initialMatch := \text{compose}_{\text{FST}}(\text{intro}(\{cb\}),$
$\qquad \text{identity}_{\text{FSA}}(\text{LiffR}(\text{FSA}_{\text{WORD}}(\langle cb \rangle), \text{xign}(D, \{cb\}))));$

After this preparation, the transducer initialMatch (Line 27) for Step 1 is defined as the composition of the free introduction of the marker † and constraining the markers in such a way that they exactly define all beginnings of a possible rewriting occurrence.

28 $$\begin{aligned}&leftToRight := \text{compose}_{\text{FST}}(\\&\quad\text{concat}_{\text{FST}}(\text{star}_{\text{FST}}(\text{concat}_{\text{FST}}(\\&\qquad\langle\text{identity}_{\text{FSA}}(\text{all}(\Sigma)),\\&\qquad\text{FST}_{\text{2WORD}}(\langle cb\rangle,\langle lb\rangle),\\&\qquad\text{identity}_{\text{FSA}}(\text{ignx}(D,\{cb\})),\\&\qquad\text{FST}_{\text{2WORD}}(\text{""},\langle rb\rangle)\rangle\\&\qquad)),\\&\quad\text{identity}_{\text{FSA}}(\text{all}(\Sigma))),\\&\quad\text{replace}_{\text{FST}}(\Sigma_x,\text{FST}_{\text{2WORD}}(\langle cb\rangle,\text{""})));\end{aligned}$$

At the second "left-to-right" step (Line 28), the stared subexpression above is used to enclose in special brackets substrings starting at a location marked by a dagger symbol † that are rewriting occurrences ("matches"). The initial dagger symbol is replaced by the left marker $\ll$. A closing $\gg$ is inserted to mark the end of the match. The first and the third identity transducer ensure that before the first matching zone and after the last matching zone we only have symbols from Σ. The second, central identity transducer still tolerates dagger symbols in matched regions. At the end also all internal dagger symbols are eliminated using a composition with an obligatory replace relation. As we mentioned above the output of the "left-to-right" step is not unique in general. Each possible output represents a left-most match decomposition of the input where matches are marked with left and right brackets. The third step selects the left-most-*longest* match decomposition.

29 $$\begin{aligned}&longestMatch := \text{identity}_{\text{FSA}}(\text{not}(\text{contain}(\\&\quad\text{concat}_{\text{FSA}}(\text{FSA}_{\text{WORD}}(\langle lb\rangle),\\&\qquad\text{intersect}_{\text{FSA}}(\text{ignx}(D,\{lb,rb\}),\\&\qquad\quad\text{contain}(\text{FSA}_{\text{WORD}}(\langle rb\rangle))\\&\qquad\quad)\\&\quad)\\&))));\end{aligned}$$

Imagine that we have marked an unit that does *not* represent a left-most-longest match. Consider the first of these units in the natural left-to-right reading order. We have marked the correct starting point, but at this position an element of D is starting that is longer than the marked unit. But then at this position we find a zone representing a unit of D (possibly with brackets of any type included), and containing at least one closing bracket from our shorter match marked. When we succeed to exclude marked texts where we find such a situation, then we know that we only have left-most-longest matches. The longestMatch constraint (Line 29) follows this strategy. The concatenation operation detects regions starting from a left marker with enriched D-zones including a shorter match. Using the not-contain operation all marked texts of this form are excluded.

30 $replacement :=$
$\text{replace}_{\text{FST}}(\Sigma_x, \text{concat}_{\text{FST}}(\langle \text{FST}_{\text{2WORD}}(\langle lb \rangle, ""), T, \text{FST}_{\text{2WORD}}(\langle rb \rangle, "")\rangle));$
31 ;

The final "replacement" step simply replaces the bracketed rewriting occurrences with the corresponding replacements defined by T using the corresponding obligatory rewrite relation. All special brackets are eliminated as well.

Remark 11.4.2 The function $\text{replaceLML}_{\text{FST}}(\Sigma, T)$ constructs the empty transducer in the case where $\varepsilon \in L_{\times 2}(T)$. This deficiency can be avoided by defining the initialMatch in this case to insert the dagger symbol at every position of the input. If needed one can easily amend the above program to cope with this case.

Remark 11.4.3 There are more sophisticated versions of rewrite rules that further involve contextual constraints. Additional constraints for the left and right context of an occurrence can further increase the expressive power of the rewrite relations. Using similar techniques as in Program 11.4.1 one can construct transducers describing rewrite relations for rules with contextual constraints, see [Kaplan and Kay, 1994, Karttunen, 1996, Gerdemann and van Noord, 1999, Hulden, 2009].

Tokenization with rewrite rules

Rewrite relations have many applications such as text highlighting and filtering, computational morphology [Kaplan and Kay, 1994, Beesley and Karttunen, 2003], shallow parsing [Gross, 1997] and others. We conclude the chapter with an application for text tokenization based on [Karttunen, 1996].

Example 11.4.4 The following program realizes a simple tokenizer for English by composing 3 rewrite transducers.

```
1  import ← "Program_11.4.1.cm";
2  Alphabet := {' ',...,'~'} ∪ {'\n','\t'};
3  whiteSpaces := {' ','\t','\n'};
4  upperCaseLetters := {'A',...,'Z'};
5  lowerCaseLetters := {'a',...,'z'};
6  Digits := {'0',...,'9'};
7  Letters := upperCaseLetters ∪ lowerCaseLetters;
8  capitalize : SYMBOL → SYMBOL;
9  capitalize(c) := { c − 32   if c ∈ lowerCaseLetters
                    { c        otherwise                ;
10 riseCase :=
       star_FST(union_FST(⟨FST_2WORD(⟨c⟩, ⟨capitalize(c)⟩) | c ∈ Alphabet⟩));
11 multiWordExpressionList :=
       ⟨"AT LEAST","HEAD OVER HEELS","IN SPITE OF",...⟩;
12 multiWordExpression :=
       union_FSA(⟨FSA_WORD(e) | e ∈ multiWordExpressionList⟩);
13 Token := union_FSA(⟨
       plus_FSA(symbolSet2FSA(Letters)),
```

```
        plus_FSA(symbolSet2FSA(Digits)),
        domain_FST(compose_FST(riseCase, identity_FSA(multiWordExpression))),
        symbolSet2FSA(Alphabet \ whiteSpaces)⟩);
14  insertLeadingNewLine :=
        concat_FST(FST_2WORD("","\n"), identity_FSA(all(Alphabet)));
15  clearSpaces := replaceLML_FST(Alphabet,
        product_FSA(plus_FSA(symbolSet2FSA(whiteSpaces)), FSA_WORD(" ")));
16  markTokens := replaceLML_FST(Alphabet,
        concat_FST(identity_FSA(Token), FST_2WORD("","\n")));
17  clearLeadingSpace := compose_FST(⟨
        insertLeadingNL,
        replace_FST(Alphabet, FST_2WORD("\n ","\n")),
        inverse_FST(insertLeadingNewLine)⟩);
18  Tokenizer := compose_FST(⟨clearSpaces,
                             markTokens,
                             clearLeadingSpace⟩);
```

In the first lines the above program defines the different sets of symbols. In Lines 8-10 the transducer *riseCase* is defined. This transducers converts any text to upper case letters without affecting the other symbols. In Line 11 the list of multiword expressions is defined. This list can contain hundreds or even thousands of entries. The automaton for the multi-word expressions is defined in Line 12. Then in Line 13 a token is defined as either as a sequences of letters, a sequence of digits, an expression representing a multi-word expression up to lower case/upper case differences, or a single symbol distinct from white space. The transducer insertLeadingNewLine (Line 14) inserts a new-line symbol in front of the leading text. The three main relations are defined as follows. First, *clearSpaces* reduces strings of white spaces to a single space (Line 15). Second, the transducer *markTokens* inserts a new-line symbol after each token (Line 16). The third transducer *clearLeadingSpace* (Line 17) deletes a space before each token, if present. This is done by inserting a leading new-line symbol, replacing each new-line symbol followed by a space with a new-line symbol. The inserted new-line symbol in front of the text is deleted. Finally, the Tokenizer is defined as the composition of the three main transducers in Line 18. Note that the use of the left-most-longest match strategy is crucial for the definition of the transducers *clearSpaces* and *markTokens*. Since the resulting transducer is functional it can be represented as a bimachine for achieving optimal efficiency of the tokenization process.

Remark 11.4.5 If used without changes, the program in Example 11.4.4 can run out of memory. In a real-world scenario one has to minimize the automata and pseudo-minimize the transducers constructed in the intermediate steps. In this way a substantial efficiency gain is achieved and the *Tokenizer* transducer can be constructed without difficulties even for large lists of multi-word expressions.

Bibliography

[Aho and Corasick, 1975] Aho, A. and Corasick, M. (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340.

[Allauzen et al., 2007] Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). Openfst: A general and efficient weighted finite-state transducer library. In Holub, J. and Žďárek, J., editors, *Implementation and Application of Automata*, pages 11–23, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Béal et al., 2003] Béal, M.-P., Carton, O., Prieur, C., and Sakarovitch, J. (2003). Squaring transducers: an efficient procedure for deciding functionality and sequentiality. *Theoretical Computer Science*, 292(1):45 – 63.

[Beesley and Karttunen, 2003] Beesley, K. and Karttunen, L. (2003). *Finite State Morphology*. CSLI studies in computational linguistics: Center for the Study of Language and Information. CSLI Publications.

[Berstel, 1979] Berstel, J. (1979). *Transductions and context-free languages*. Leitfäden der angewandten Mathematik und Mechanik. Teubner.

[Björklund et al., 2014] Björklund, H., B., and J., Zechner, N. (2014). Compression of finite-state automata through failure transitions. *Theoretical Computer Science*, 557:87–100.

[Carrasco and Forcada, 2002] Carrasco, R. C. and Forcada, M. L. (2002). Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2):207–216.

[Choffrut, 1977] Choffrut, C. (1977). Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationelles. *Theoretical Computer Science*, 5:325–338.

[Crochemore and Hancart, 1997] Crochemore, M. and Hancart, C. (1997). Automata for matching patterns. In *Handbook of formal languages*, pages 399–462. Springer.

[Daciuk et al., 2000] Daciuk, J., Mihov, S., Watson, B., and Watson, R. (2000). Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16.

[Daciuk and Weiss, 2012] Daciuk, J. and Weiss, D. (2012). Smaller representation of finite state automata. *Theor. Comput. Sci.*, 450:10–21.

[Eilenberg, 1974] Eilenberg, S. (1974). *Automata, Languages, and Machines - Volume A*, volume volume 59 of Pure and Applied Mathematics. Academic Press.

[Eilenberg, 1976] Eilenberg, S. (1976). *Automata, Languages, and Machines - Volume B*. Academic Press.

[Friedl, 1997] Friedl, J. (1997). *Mastering Regular Expressions*. O'Reilly & Associates.

[Gerdemann and van Noord, 1999] Gerdemann, D. and van Noord, G. (1999). Transducers from rewrite rules with backreferences. In *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics (EACL 99)*, pages 126–133.

[Gerdjikov, 2018] Gerdjikov, S. (2018). A general class of monoids supporting canonisation and minimisation of (sub)sequential transducers. In *Language and Automata Theory and Applications - 12th International Conference, LATA 2018, Ramat Gan, Israel, April 9-11, 2018, Proceedings*, pages 143–155.

[Gerdjikov and Mihov, 2017a] Gerdjikov, S. and Mihov, S. (2017a). Myhill-nerode relation for sequentiable structures. *CoRR*, abs/1706.02910.

[Gerdjikov and Mihov, 2017b] Gerdjikov, S. and Mihov, S. (2017b). Over which monoids is the transducer determinization procedure applicable? In *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, pages 380–392.

[Gerdjikov et al., 2017] Gerdjikov, S., Mihov, S., and Schulz, K. U. (2017). A simple method for building bimachines from functional finite-state transducers. In Carayol, A. and Nicaud, C., editors, *Implementation and Application of Automata*, pages 113–125. Springer International Publishing.

[Gross, 1997] Gross, M. (1997). The construction of local grammars. In Roche, E. and Schabes, Y., editors, *Finite-State Language Processing*, pages 329–354. MIT Press.

[Hopcroft et al., 2006] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Pearson. 3nd edition.

[Hulden, 2009] Hulden, M. (2009). *Finite-State Machine Construction Methods and Algorithms for Phonology and Morphology*. PhD thesis, University of Arizona.

[Hutton, 2007] Hutton, G. (2007). *Programming in Haskell*. Cambridge University Press.

[Kaplan and Kay, 1994] Kaplan, R. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–279.

[Karttunen, 1996] Karttunen, L. (1996). Directed replacement. In Joshi, A. and Palmer, M., editors, *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 108–115, Santa Cruz.

[Karttunen, 1997] Karttunen, L. (1997). The replace operator. In Roche, E. and Schabes, Y., editors, *Finite-State Language Processing*, pages 117–147. MIT Press.

[Karttunen et al., 1997a] Karttunen, L., Chanod, J.-P., Grefenstette, G., and Schiller, A. (1997a). Regular expressions for language engineering. *Journal of Natural Language Engineering*, 2(4):307–330.

[Karttunen et al., 1997b] Karttunen, L., Gaál, T., and Kempe, A. (1997b). Xerox finite-state tool. Technical report, Xerox Corporation.

[Kourie et al., 2012] Kourie, D., Watson, B., Cleophas, L., and Venter, F. (2012). Failure deterministic finite automata. In *Proceedings of Prague Stringology Conference*, pages 28–41.

[Kozen, 1997] Kozen, D. C. (1997). *Automata and Computability*. Springer, New York, Berlin.

[Lewis and Papadimitriou, 1998] Lewis, H. R. and Papadimitriou, C. H. (1998). *Elements of the Theory of Computation*. Prentice-Hall, Upper Saddle River, New Jersey. 2nd edition.

[Maurel and Guenthner, 2005] Maurel, D. and Guenthner, F. (2005). *Automata and Dictionaries*. Texts in Computer Science. College Publications.

[Mehryar Mohri, 2015] Mehryar Mohri, Michael Riley, A. T. S. (2015). Automata and graph compression.

[Mihov, 1998a] Mihov, S. (1998a). Direct building of minimal automaton for given list. *Annuaire de l'Université de Sofia "St. Kl. Ohridski"*, 91, livre 1.

[Mihov, 1998b] Mihov, S. (1998b). Direct construction of minimal acyclic finite states automata. *Annuaire de l'Université de Sofia "St. Kl. Ohridski"*, 92, livre 2.

[Mihov and Maurel, 2001] Mihov, S. and Maurel, D. (2001). Direct construction of minimal acyclic subsequential transducers. In *Proceedings of the Conference on Implementation and Application of Automata CIAA'2000*, number 2088 in LNCS, pages 217–229. Springer.

[Mohri, 1995] Mohri, M. (1995). Matching patterns of an automaton. In *Combinatorial Pattern Matching*, pages 286–297. Springer.

[Mohri, 1996] Mohri, M. (1996). On some applications of finite-state automata theory to natural language processing. *Journal of Natural Language Engineering*, 2:1–20.

[Mohri, 1997] Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

[Mohri, 2000] Mohri, M. (2000). Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234:177–201.

[Mohri, 2004] Mohri, M. (2004). Weighted finite-state transducer algorithms. an overview. In *Formal Languages and Applications*, pages 551–563. Springer.

[Mohri et al., 2008] Mohri, M., Pereira, F., and Riley, M. (2008). Speech recognition with weighted finite-state transducers. In *Springer Handbook of Speech Processing*, pages 559–584. Springer.

[Mohri and Sproat, 1996] Mohri, M. and Sproat, R. (1996). An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Meeting of the Association for Computational Linguistics (ACL'96)*, pages 231–238, Santa Cruz, CA.

[Navarro and Raffinot, 2002] Navarro, G. and Raffinot, M. (2002). *Flexible Pattern Matching in Strings*. Cambridge University Press, Cambridge, UK.

[P.Pisolkar and Lahane, 2015] P.Pisolkar, U. and Lahane, S. R. (2015). A survey on deterministic finite automata compression techniques. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 4.

[Pratt and Zelkowitz, 2000] Pratt, T. W. and Zelkowitz, M. V. (2000). *Programming Languages: Design and Implementation (4td Ed.)*. Pearson.

[Reutenauer and Schützenberger, 1991] Reutenauer, C. and Schützenberger, M. P. (1991). Minimization of rational word functions. *SIAM J. Computing*, 20(4):669–685.

[Roche and Schabes, 1997a] Roche, E. and Schabes, Y. (1997a). Deterministic part-of-speech tagging with finite-state transducers. In Roche, E. and Schabes, Y., editors, *Finite-State Language Processing*, Language, Speech, and Communication, pages 205–240. The MIT Press.

[Roche and Schabes, 1997b] Roche, E. and Schabes, Y. (1997b). Introduction. In Roche, E. and Schabes, Y., editors, *Finite-State Language Processing*, pages 1–66. MIT Press.

[Sakarovitch, 2009] Sakarovitch, J. (2009). *Elements of Automata Theory*. Cambridge University Press, New York, NY, USA.

[Schmid, 2006] Schmid, H. (2006). A programming language for finite state transducers. In Yli-Jyrä, A., Karttunen, L., and Karhumäki, J., editors, *Finite-State Methods and Natural Language Processing*, pages 308–309, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Schulz and Mihov, 2002] Schulz, K. U. and Mihov, S. (2002). Fast String Correction with Levenshtein-Automata. *International Journal of Document Analysis and Recognition*, 5(1):67–85.

[Schützenberger, 1961] Schützenberger, M.-P. (1961). A remark on finite transducers. *Information and Control*, 4:185–196.

[Schützenberger, 1975] Schützenberger, M. P. (1975). Sur les relations rationnelles. In *GI Conference on Automata Theory and Formal Languages*, volume 33 of *Springer LNCS*, pages 209–213.

[Schützenberger, 1977] Schützenberger, M.-P. (1977). Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4:47–57.

[Schwartz et al., 1986] Schwartz, J. T., Dewar, R. B., Schonberg, E., and Dubinsky, E. (1986). *Programming with Sets; an Introduction to SETL*. Springer-Verlag, Berlin, Heidelberg.

[Skut, 2003] Skut, W. (2003). Incremental construction of minimal sequential transducers. In *Computational Linguistics in the Netherlands 2003, December 19, Centre for Dutch Language and Speech, University of Antwerp.*

[van Noord, 2000] van Noord, G. (2000). Treatment of epsilon moves in subset construction. *Comput. Linguist.*, 26(1):61–76.

Index

e-closure, 34
e-free monoidal finite-state automaton, 34
k-ary function, 6
n-ary string relation, 12
n-tape letter automaton, 70
n-tuple, 3
n-way concatenation of words, 11
n-way reversal, 12
n-way reverse relation, 13
w-successor of state, 26

Active states, 26
Admissible advance, 78
Advance function, 76
Alphabet, 9
Antisymmetric relation, 4

Backward e-closure, 34
Backwards f-automaton, 217
Backwards failure deterministic finite-state automaton, 217
Bijective function, 7
Bimachine, 121
Binary regular string relation, 75
Bounded variation property, 89

Canonical form of transducer, 109
Cardinality, 3
Cartesian product, 3
Cartesian product of automata, 45
Cartesian product of monoids, 17
Classical n-tape automaton, 69
Classical automaton language, 25
Classical bimachine, 121
Classical finite state transducer , 74
Classical finite-state automaton, 22
Classical letter transducer, 74
Classical regular expression, 31
Classical regular language, 30
Classical subsequential transducer, 87
Cloning a state, 233
Codomain, 6
Coloured automaton, 55
Colouring of words, 55
Completed transition function of f-automaton, 217
Composition of relations, 4
Concatenation Kleene start for string relation, 13
Concatenation of n-tuples of words, 11
Concatenation of classical languages, 10
Concatenation of string relations, 12
Concatenation of words, 9
Convergent state, 224

Depth of a trie state, 41
Deterministic finite-state automaton, 39
Deterministic finite-state transducer, 85
Difference of finite-state automata, 44
Domain, 6

Edge of bimachine, 124
Empty word, 9
Equivalence class, 5
Equivalence relation, 5
Equivalent coloured automata, 55
Equivalent monoidal finite-state automata, 25
Equivalent states, 26
Equivalent states of coloured automaton, 57
Equivalent states of subsequential transducer, 87
Expanded output automaton of transducer, 114

f-automaton, 217
Failure deterministic finite-state automaton, 217
Forward e-closure, 34

Free companion of monoidal automaton, 27
Free monoid, 10
Function represented by a bimachine, 122
Function represented by a subsequential transducer, 87
Function represented by functional transducer, 73
Functional composition, 7
Functional transducer, 73
Functionalizing a relation, 6

Generalized completed transition function of f-automaton, 217
Generalized output function of a bimachine, 123
Generalized projection of a relation, 5
Generalized transition, 25
Generalized transition function, 40
Generalized transition relation, 25

Homomorphic image, 16
Homomorphic image of automaton, 27

Image under a function, 6
Image under a relation, 4
Index of equivalence relation, 5
Infinitely ambiguous relation, 4
Infinitely ambiguous transducer, 73
Infix, 9
Infix occurrence, 10
Injective function, 7
Input language of transducer, 86
Intersection of finite-state automata, 44
Inverse relation, 4
Isomorphic automata, 26
Isomorphic coloured automata, 56
Iterated advance function, 76

Kernel, 52
Kleene star, 11
Kleene star (relational), 5

Language, 10
Language of a state, 25
Language of coloured automaton, 55
Language of f-automaton, 217
Language recognized/accepted by an automaton, 24
Left profile function of bimachine, 131
Letter transducer, 73
Linear order, 6
Longest common output, 104
Longest common prefix, 9
Loop path, 24

Maximal state output, 109
Minimal except for a word, 224, 238
Monoid, 14
Monoid homomorphism, 16
Monoid isomorphism, 16
Monoid unit element, 14
Monoidal n-tape automaton, 66
Monoidal n-tape automaton relation, 68
Monoidal automaton language, 25
Monoidal bimachine, 121
Monoidal finite-state automaton, 21
Monoidal finite-state transducer, 73
Monoidal Kleene star, 16
Monoidal language, 15
Monoidal letter transducer, 73
Monoidal product, 15
Monoidal regular n-relation, 31
Monoidal regular expression, 31
Monoidal regular language, 30
Monoidal subsequential transducer, 86
MSA, 21
Multi-tape automaton, 66
Myhill-Nerode, 48
Myhill-Nerode automaton, 56
Myhill-Nerode relation, 48, 56, 103
Myhill-Nerode transducer, 107

Null path, 24

Obligatory rewrite relation, 247
Occurrence of an infix, 10
Optional rewrite relation, 246
Output automaton of transducer, 113
Output function for transducer state, 87
Output function of bimachine, 122
Output function of subsequential transducer, 87
Output function of transducer, 86
Overlap of words, 248

Pair of successful paths, 122
Partial function, 6

Partial order, 6
Path label for a trie state, 41
Permutation, 7
Position in a string, 10
Prefix, 9
Product of monoidal languages, 15
Projection, 5
Projection of a relation, 5
Proper path, 24
Proper prefix/suffix, 9
Pseudo-deterministic transducer, 60
Pseudo-minimal bimachine equivalent a bimachine, 132

Rational string function, 75
Reachable state, 24
Real-time transducer, 74
Refinement of equivalence relation, 5
Reflexive and transitive closure, 5
Reflexive relation, 4
Regular expression, 31
Regular language, 30
Regular string function, 75
Relation, 4
Relational composition, 4
Reverse automaton, 45
Reverse function for words, 9
Reverse language, 11
Right invariant equivalence relation, 46

Sequential distance of words, 89
Set-lifted n-way concatenation, 12
Set-lifted version of a function, 7
Set-lifted version of monoid, 15
Squared output transducer, 77
State output function, 87
String, 9
String function, 14
Submonoid, 16
Subsequential finite-state transducer with initial output, 104
Subsequential function, 89
Subsequential transducer, 87
Successful path, 24
Suffix, 9
Surjective function, 7
Symmetric relation, 4

Total bimachine, 125
Total function, 6
Total monoidal subsequential transducer, 87
Transducer, 73
Transition, 22
Transition function, 85
Transition relation, 85
Transitive closure, 5
Transitive relation, 4
Trie, 40
Trimmed monoidal finite-state automaton, 33
Tuple-lifted version of a function, 8
Two-sided dictionary, 237

Unambiguous transducer, 129
Underlying automaton of monoidal subsequential transducer, 87
Underlying finite-state automaton, 73

Word, 9